View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by E-resource repository of the University of Latvia

University of Latvia
Faculty of Computing

DMITRIJS RUTKO

FUZZIFIED GAME TREE SEARCH

Doctoral Thesis

Area: Computer Science
Sub-Area: Data Processing Systems and Computer Networks

Scientific Advisor:
Dr.sc.comp., Prof.
Guntis Arnicāns

Riga 2012

The research has been supported by the European Social Fund within the project "Support for Doctoral Studies at University of Latvia".

ABSTRACT

The thesis is dedicated to the research of game tree search algorithms. It shows that exact game tree evaluation is not required to find the best move. Therefore, pruning techniques may be applied earlier, resulting in a faster search and greater performance.

A new approach called Fuzzified game tree search is proposed, which allows faster determination of the best move whilst visiting fewer nodes. Experimental results in real domain games showed an increase of 10% in performance over existing algorithms.

We also present approximation-based implementations of the Fuzzified game tree search algorithm. The paradigm of the algorithm allows us to find nearly optimal solutions efficiently, so that the "target quality" of the search can be chosen with arbitrary precision. Our experimental results showed that this kind of approximation in games could be an acceptable trade-off, demonstrating a 15% speed increase without significantly affecting the overall playing strength of the algorithm.

PREFACE

This thesis involves research performed by the author and reflected in the following publications:

1. Dmitrijs Rutko, Fuzzified Game Tree Search - Precision vs. Speed, *PRICAI 2012: Trends in Artificial Intelligence, Springer, 2012, LNAI, Volume 7458, pp. 504–515.* [Indexed by Scopus]

2. Dmitrijs Rutko, Fuzzified Tree Search in Real Domain Games, *Advances in Artificial Intelligence, Springer, 2011, LNAI, Volume 7094, pp. 149–161. Received* ***"Best Paper Award".*** [Indexed by ISI, Scopus]

3. Dmitrijs Rutko, Fuzzified Algorithm for Game Tree Search with Statistical and Analytical Evaluation. *Scientific Papers, University of Latvia, 2011, Vol. 770, pp. 90–111*

4. Dmitrijs Rutko, Fuzzified Algorithm for Game Tree Search. *Proceedings of Second Brazilian Workshop of the Game Theory Society, BWGT 2010*

In addition, the following papers not directly related to the thesis topic have also been produced in part by the author:

1. Vadim Zuravlyov, Anton Matrosov, Dmitrijs Rutko, Behavior Pattern Simulation of Freelance Marketplace*, Advances in Intelligent and Soft Computing, Springer, 2012, Volume 157, pp. 157-164*. Received ***"PAAMS '12 Award of Scientific Excellence".*** [Indexed by ISI, Scopus]

2. Dmitrijs Rutko, Vadim Zuravlyov, Anton Matrosov, Freelance Resource Management System Optimization, *Proceedings of The 2nd International Conference on Computer and Management, Wuhan, China, CAMAN 2012, IEEE Xpress, ISBN 978-1-4577-1137-4, pp. 804–807*

The results of the thesis were presented by the author at the following international conferences and workshops:

1. 12th Pacific Rim International Conference on Artificial Intelligence, PRICAI 2012, Kuching, Sarawak, Malaysia, Presentation "*Fuzzified Game Tree Search - Precision vs Speed"*

2. 4th PhD Symposium, Knowledge Technology Week, KTW 2012, Kuching, Sarawak, Malaysia, Presentation *"Finding Optimal Strategies in Games"*

3. 10th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2012), Salamanca, Spain. Presentation *"Behavior Pattern Simulation of Freelance Marketplace"*. Received ***"PAAMS '12 Award of Scientific Excellence"***

4. The 2nd International Conference on Computer and Management, Wuhan, China, CAMAN 2012, Presentation *"Freelance Resource Management System Optimization"*

5. 10th Mexican International Conference on Artificial Intelligence, Puebla, Mexico, 2011, Presentation *"Fuzzified Tree Search in Real Domain Games". Received* ***"Best Paper Award"***

6. 3rd Joint World Latvian Scientists Congress, Riga, Latvia, 2011, Section of computer science and information technology, Poster *"Optimal Strategies in Games"*

7. 7th Baltic Summer School in Technical Informatics and Information Technology, BASOTI, Riga, Latvia, 2011, Presentation *"Fuzzified Tree Search in Real Domain Games"*

8. LU and LMT Computer Science Days, University of Latvia, Ratnieki, Latvia, 2011, Presentation "*Uncertain Reasoning in Games*"

9. Innovations in Algorithmic Game Theory, Hebrew University of Jerusalem, Israel, 2011, Poster "*Fuzzified Tree Search in Real Domain Games*"

10. 6th Pan Pacific Conference on Game Theory, Tokyo Institute of Technology, Japan, 2011, Presentation "*Analytical Game Tree Evaluation for Fuzzified Search Algorithm*"

11. Algorithmic Game Theory, IPAM / UCLA, Los Angeles, USA, 2011, Poster "*Optimized Search in Games*"

12. Joint Estonian-Latvian Theory Days, Rakari, Latvia, 2010, Presentation "*Applied Machine Learning in Game Theory*"

13. Second Brazilian Workshop of the Game Theory Society, Sao Paolo, Brazil, 2010, Presentation "*Fuzzified Algorithm for Game Tree Search*"

14. International Summer School on Algorithmic Game Theory, Shanghai, China, 2010, Presentation "*Fuzzified Algorithm for Game Tree Search*"

15. LU 67. Conference, Riga, Latvia, 2009, Section of computer science and information technology, Presentation "*Game Tree Search*"

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Prof. Guntis Arnicāns, whose support and ideas were one of the key factors for me to pursue and complete the research.

In addition, I am thankful to Assoc. Prof. Jānis Zuters, who by his example encouraged me to perform the extended research in the area of the game theory and to Nikolajs Nahimovs for sharing valuable ideas and concepts.

I truly thank my parents and family whose support was critical to me for writing the thesis.

Let me also express my gratitude to other colleagues, friends and relatives who have helped me while working on this research.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

1.1 Relevance of the Topic

Artificial Intelligence (AI) is one of the active research directions of the modern computer science. Ideas on the use of AI for solving complex challenges have appeared long time ago. Today, it is increasingly difficult to achieve the technical progress by using automated repeating operations only. Applications need to be capable of self-learning and self-organisation, to generalize the information obtained through their experience. AI helps people to facilitate thinking and decision-making processes in many situations where there is a need for very deep analysis of the specific factors and the requirement to investigate a lot of different possible solutions to the problem.

Game theory is one of the classical Artificial Intelligence sub-fields. It focuses on optimal strategies for decision-making processes involving two or more players. Each player has its purpose, and each of the players use a certain strategy, which guarantees certain outcome depending on the other players' actions selected. Game theory describes the optimal strategy, given knowledge of the opponents, available resources and possible actions in particular situations.

One of current research interests is the ancient oriental game of Go that has long been considered a grand challenge for Artificial Intelligence. For decades, computer Go has deed the classical methods in game tree search that worked so successfully for chess and checkers. However, recent play in computer Go has been transformed by a new paradigm for tree search based on Monte-Carlo methods. Programs based on Monte-Carlo tree search now play at human-master levels and are beginning to challenge top professional players [48].

Over the last few years, there has been explosive growth in the research done at the interface of computer science, game theory, and economic theory, largely motivated by the emergence of the Internet. Algorithmic Game Theory develops the central ideas and results of this new and exciting area [46].

However, the use of game theory is not limited to theoretical computer science problems. It is used in real life, such as the analysis of economic processes (consumer response to changes in the market), business (prevention of abuse activities in

auctions), political science (choosing corresponding position in the international negotiation process), the modern computerized auctions lead by the rapid development of search engines, social grids and other heavily used applications. Game theory also extends to many other industries, such as the expert's decision-making, the court action, the military strategic planning and marketing technology.

Game theory is being used, for example, in biology to explain many seemingly incongruous phenomena in nature. Maynard Smith writes, "Paradoxically, it has turned out that game theory is more readily applied to biology than to the field of economic behaviour for which it was originally designed". It laid the basis for the development of a new direction – Evolutionary Game Theory [47].

1.2 Objectives of the Research

The overall objective of the research is to improve AI accuracy of solving complex problems. The thesis is aimed specifically at achieving a higher level of play in logic games.

The thesis studies game theory aspects and performs analysis of game tree search algorithms. It has the following tasks:

- Analyse characteristics of game tree search algorithms and identify its strengths and weaknesses;
- Improve existing search techniques;
- Find new more efficient search techniques;
- By using the new principles and ideas create algorithms that work faster and thus able to find the optimal solution even if all options cannot be analysed.

1.3 Overview of the Results

The thesis summarizes research results in the following directions:

- We propose a new approach to search in games – Fuzzified game tree search, and have developed corresponding search algorithm [12];
- We propose two enhancements for Fuzzified search based on game tree statistical self-learning and analytical evaluation in order to find the optimal control parameters required by the algorithm [13];

- We implement given algorithm in a real game "Hey! That's My Fish!", resulting in a 10% performance improvement when compared with other known algorithms [43];
- We show that the approximation paradigm of Fuzzified search algorithm can be used to search for the best move with certain accuracy. The proposed version gave 15% performance improvement without significantly affecting the overall level of play of the program [44].

1.4 Foreword

Games have attracted the intellectual resources of humankind for a long time. For researchers of Artificial Intelligence, the nature of games is a challenging subject for study. Usually, it is easy to represent the state of a game but remarkably hard to solve it; which is also true in the real world, where finding optimal solutions is often infeasible.

Games are usually represented in their extensive form with the help of a game tree, which starts at the initial position and contains all the possible subsequent moves from each position. Classical game tree search algorithms, such as Minimax and Negamax, operate by using a complete scan of all the nodes of the game tree and are considered too inefficient. The most practical approaches are based on the Alpha-Beta pruning technique, which seeks to reduce the number of nodes to be evaluated in the search tree. It stops the evaluation of a move completely, if at least one possibility is found where the current move is proven worse than the previously examined move and thus, such moves do not need to be evaluated further.

More advanced extensions and additional improvements of Alpha-Beta are known as PVS, NegaScout and NegaC*. Another group of algorithms like SSS*/Dual* and MTD(f) is based on the Best-First Search in a game tree. Potentially, they could be even more efficient; however, they typically require high memory consumption.

We study two-player zero-sum games and our research is focused mainly on game tree search algorithms. Through analysing and comparing these algorithms, it can be seen that in many cases, the decision about the best move can be made before the exact game tree Minimax value is obtained. A new approach has been proposed that allows the best move to be found more quickly whilst visiting fewer nodes.

We also analyse different approaches for selective searches and present an approximation-based paradigm that allows us to find nearly optimal solutions efficiently, so that the "target quality" of the search can be chosen with arbitrary precision. It allows us to find the right balance between precision and speed during the search.

1.5 Structure of the Thesis

The thesis is organised as follows.

Part I contains the introduction and detailed review of previous relevant work in the area of game tree search:

- The 2nd chapter “Game Theory Concepts” provides an introduction to game theory, core concepts and basic algorithms.
- In the 3rd chapter “Search Algorithms”, the current status in the area of game tree search algorithms is discussed.
- In the 4th chapter “Selective Search”, the probabilistic forward pruning and state-of-the-art algorithms are described.

Part II contains a full description and the main results of the author’s research work:

- In the 5th chapter “Fuzzified Tree Search”, the idea that allows the game tree search to be performed in a manner based on the move that leads to the best result is proposed. Following this, the algorithm structure and implementation details are explained.
- In the 6th chapter “Fuzzified Tree Search in Real Domain Games”, the experimental setup and empirical results on the search performance obtained in a real domain are shown.
- In the 7th chapter “Fuzzified Tree Search - Precision vs. Speed”, a new enhancement to the Fuzzified Tree Search algorithm based on the quality of the search is presented. This allows us to adjust both the target quality and performance to suit our needs.

The thesis is concluded with thoughts on future research directions.

PART I

INTRODUCTION AND RELEVANT WORK

2 GAME THEORY CONCEPTS

2.1 Introduction to Game Theory

Game theory studies multi-player environment and internal interaction processes as a game regardless of whether players cooperate or compete. In Artificial Intelligence, most varieties of games are of a special kind – two player zero-sum games with perfect information. This means that these games are:

1) *deterministic* – all possible moves are determined from the given position only;
2) *perfect information* – all information about game state is available to both players;
3) moves are performed by the players sequentially and alternately;
4) the result of the game is equal and opposite for the players.

For example, if one player wins in a game of chess, then the other player necessarily loses with the same points.

The structure of a game could be defined formally as a search problem (a computational problem that requires the identification of a solution from some, possibly infinite, solution space (set of possible solutions) [31]) containing the following elements:

- $S(0)$ – initial state, which specifies how the game is set up at the start;
- $Players(s)$ – defines which player has a move in the current state "s";
- $Actions(s)$ – returns a set of legal moves in the state "s";
- $Result(s, a)$ – transition model, which defines the result of move "a" in the state "s";
- $Terminal\_Test(s)$ – a terminal test, which notifies when the game is over;
- $Terminal\_States()$ – a set of the states, where the game has ended;
- $Utility(s, p)$ – utility function (also could be called objective function or payoff function), which defines the final numeric value of a game that ends in the terminal state "s" for a player "p".

Usually, games are represented in their extensive form with the help of a game tree, which starts at the initial position and contains all the possible moves from each position. Figure 1 shows a part of the game tree for Tic-Tac-Toe (noughts and crosses).

From the initial state, the Max(X) player has nine possible moves and the Min(O) player has the next eight possible moves, etc. For this particular game search, the tree is relatively small – smaller than 9! = 362 880 terminal nodes. However, for chess there are over 10^{40} nodes [11] and thus, it is not feasible to construct the whole tree.

Usually, moves are ordered and a single move by one player is called a *ply*.

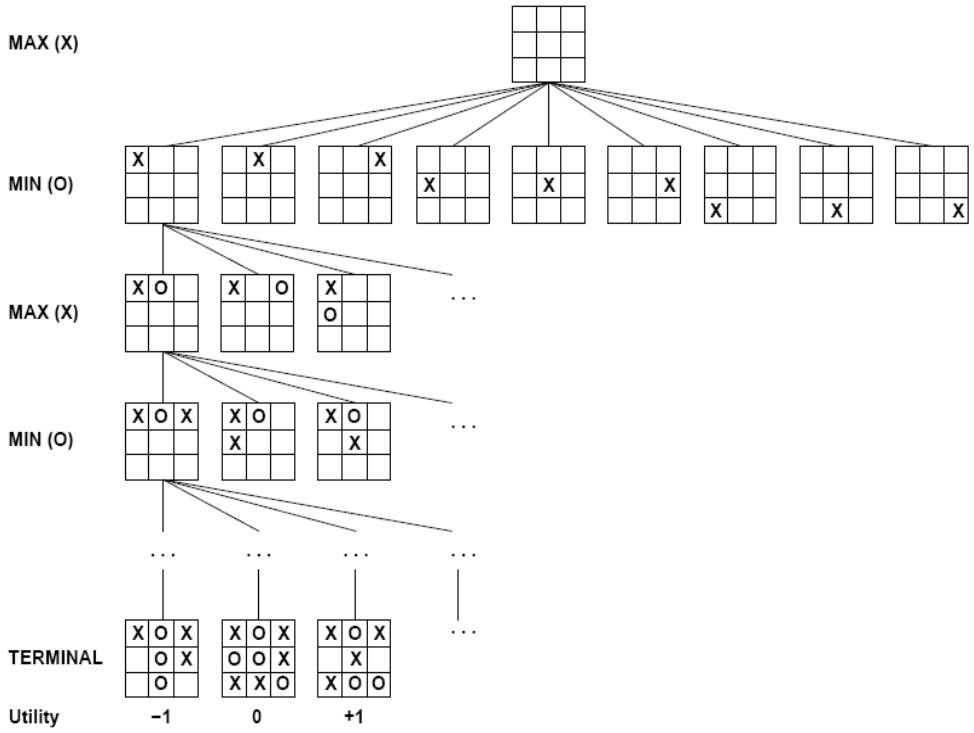


Figure 1. Game tree representing Tic-Tac-Toe [11]

2.2 Optimal Strategies in Games

Optimal strategy is the sequence of actions of a player that leads to a desired state – a terminal state that is a win. However, the two players have opposite goals and this should be taken into account while searching for an optimal move. In others words, an optimal strategy guarantees an outcome that is not worse than that given by any other strategy.

Given a game tree, the optimal strategy can be obtained from the *minimax value* of each node, which could be calculated with a function Minimax(s). The Minimax value of a node is simply its utility; otherwise, it is the utility of the corresponding node. Therefore, it could be written in the following form:

$$Minimax(s) =$$

$$\begin{cases} Utility(s) & if\ Terminal\_Test(s) \\ \max a \in Actions(s)\ Minimax(Result(s,a)) & if\ Player(s) = MAX \\ \min a \in Actions(s)\ Minimax(Result(s,a)) & if\ Player(s) = MIN \end{cases} \quad (1)$$

This definition of Minimax assumes that both players, MAX and MIN, play optimally. If either player deviates from an optimal strategy, then the opponent gains even more advantage.

The Minimax algorithm performs a complete depth-first scan of the game tree. If we note the maximum depth of the tree as d and there are w legal moves at each point (game width), then the time complexity of the algorithm is

$$Time\ complexity = O(w^{\mathrm{d}}) \quad (2)$$

and the space complexity is

$$Space\ complexity = O(wd) \quad (3)$$

For real games, this time complexity is completely impractical but this algorithm serves as a basis for further improvements and comparisons.

2.3 Multiplayer Games

Many popular games involve more than two players. Let us examine the idea in more detail.

The main difference is that we need to replace the single value evaluation of a node with a *vector of values*. For example, in a three-player game, where players are denoted as A, B and C, respectively, the vector (V_a, V_b, V_c) will be associated with each node. For terminal states, it will give the utility of the state from the point of view of each player. The simplest way to implement this idea is to have the utility function return an array of utilities.

However, multiplayer games tend to involve *alliances*, which could be formal or informal and made or broken during the game. Therefore, dealing with these cooperative strategies adds additional challenges to multiplayer games.

2.4 Alpha-Beta Pruning

Classical game tree search algorithms, such as Minimax, operate using a complete scan of all the nodes of the game tree and are considered too inefficient. The most practical approaches are based on the Alpha-Beta pruning technique [29].

Alpha-Beta is a search algorithm that tries to reduce the number of nodes to be evaluated in the search tree by the Minimax algorithm. It completely stops evaluating a move when at least one possibility has been found that proves the move worse than a previously examined one; such moves do not need to be evaluated further. When applied to a standard Minimax tree, it returns the same move as Minimax would but prunes away branches that cannot possibly influence the final decision [57].

An illustration of the Alpha-Beta approach is given in Figure 2.

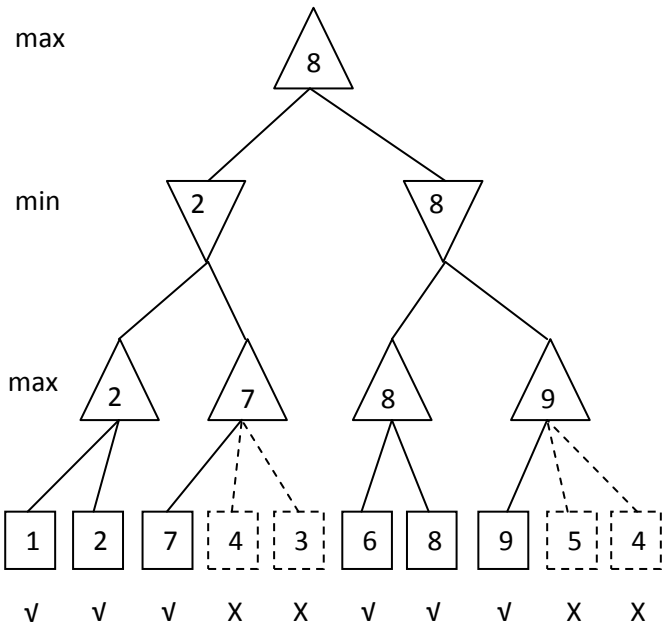


Figure 2. The Alpha-Beta approach

The game tree in Figure 2 has two branches with Minimax values 2 and 8 for the left and right sub-trees, respectively. In order to find the best move, the Alpha-Beta algorithm scans all the sub-trees from left to right and is forced to evaluate almost every node. The possible cut-offs are depicted with a dashed line (at each step, the previous evaluation is smaller than the value of the currently checked node).

When all the nodes are checked, the algorithm compares the top-level sub-trees. The evaluation of the left and the right branches are 2 and 8, respectively; the highest outcome is chosen, and the best move goes to the right sub-tree.

The main benefit of Alpha-Beta pruning is that many branches of the search tree can be eliminated. Thus, the search can be limited to a 'more promising' sub-tree and a deeper search can be performed in the same amount of time.

2.5 Move Ordering

The effectiveness of Alpha-Beta pruning is highly dependent on the move order in which the nodes are examined. For example, in Figure 2, if we scan nodes at the max level in reverse order {9, 8, 7, 2} successful pruning cannot be applied. This implies that Alpha-Beta pruning is only effective when the best move is examined first.

The main idea behind move ordering is an attempt to examine those moves first that have the highest probability of being the best moves available. If this could be achieved, then the Alpha-Beta search needs only

$$O(w^{\mathrm{d}/2}) \tag{4}$$

nodes in the best case scenario to find the best move, instead of $O(w^{\mathrm{d}})$ nodes as required by Minimax [29]. This means that the effective branching factor becomes $\sqrt{w}$ instead of w. For example, in a chess game with on average 35 moves available in each position, the new effective branching factor becomes 6. In other words, we can say that Alpha-Beta allows us to search the game tree two times deeper in the same amount of time compared with Minimax. However, inappropriate move ordering may significantly affect the efficiency of the search algorithm and potentially result in time complexity equal to that of Minimax.

2.6 Iterative Deepening

Adding dynamic move ordering techniques that for example, rely on moves found in the past, brings us close to theoretical limits [11].

Because the Minimax algorithm and its variants are inherently depth-first, strategies such as *iterative deepening* are usually used in conjunction with Alpha-Beta, so that a

reasonably good move can be returned even if the algorithm is interrupted before it has finished execution. Iterative deepening runs repeatedly, increasing the depth limit with each iteration until it reaches d, the depth of the shallowest goal state. Another advantage of using iterative deepening is that searches at shallower depths give hints about move ordering, which can help produce cut-offs for higher-depth searches much earlier than would otherwise be possible [57].

2.7 Transposition Tables

Transposition tables are another technique used to accelerate the search of the game tree in computer chess and other computer games. In many games, it is possible to reach a given position, which is called transposition, in more than one way. In general, after two moves there are four possible transpositions because either player may swap their move order. Therefore, it is still likely that the program will end up analysing the same position several times. To avoid this problem transposition tables store previously analysed positions of the game [30].

The use of a transposition table can have a dramatic effect, sometimes doubling the reachable search depth in chess. On the other hand, if we evaluate a million nodes per second, at some point it is not practical to keep all of them in the transposition table. Various strategies have been implemented to choose which nodes to keep and which to discard [11].

2.8 Evaluation Functions

The Minimax algorithm scans the entire game tree, whereas the Alpha-Beta algorithm allows the pruning of a large part of it. However, both algorithms have to examine all paths of the search space in order to reach the terminal state (leaf node) when the utility function could be applied. Even though this search depth is not affordable because in most real games moves should be made in reasonable time – from several seconds to several minutes at most.

Claude Shannon in the paper “Programming a Computer for Playing Chess” proposed a different approach [49]. Instead of a full search up to the game end, programs should terminate their search earlier and apply a *heuristic evaluation function*

to states (nodes), which means that non-terminal nodes should be transformed into terminal ones (leaves). In other words, the suggestion is to change the Minimax and Alpha-Beta algorithms in two ways, i.e., replace the utility function by a heuristic evaluation function EVAL, which estimates the utility of the position and replaces the terminal test by a cut-off test that decides when to apply EVAL. That gives us the following equation for a heuristic Minimax for state s and maximum depth d [11]:

$$H\_Minimax(s,d) = \begin{cases} EVAL(s) & if\ Cutoff\_Test(s,d) \\ \max a \in Actions(s)\ H\_Minimax(Result(s,a), d+1) & if\ Player(s) = MAX \\ \min a \in Actions(s)\ H\_Minimax(Result(s,a), d+1) & if\ Player(s) = MIN \end{cases} \quad (5)$$

The evaluation function returns an estimate of the expected utility of a given position in a game. The idea proposed by Shannon was not new. For centuries chess players had been developing ways to analyse board positions because modern computers are more efficient in pure search. It is obvious that the performance of a game-playing program greatly depends on the quality of its evaluation function. An inaccurate evaluation function may lead an agent to a position, which will turn out to be a lost one.

Therefore, the following conditions should be considered when designing a good evaluation function [11]:

- State ordering produced by an evaluation function should correspond to the ordering of the true utility function. States ranked higher by the utility function should also be higher in the ranking provided by the evaluation function.
- The computation of the evaluation function should be fast; otherwise, it cannot be used efficiently in search algorithms.
- The results of the evaluation function for intermediate (non-terminal) states should strongly correlate with actual chances of winning.

A typical mathematical representation of an evaluation function for chess would be a weighted linear function expressed as:

$$EVAL(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \quad (6)$$

where w_i is the weight and f_i is the feature of the position. For chess, f_i could be the number of each kind of piece on the board (number of pawns, bishops, knights, queens, etc.) and w_i could be the values of the pieces (1 for pawn, 3 for bishop, 5 for rook, 9 for queen, etc.).

However, this representation assumes that features and their values are independent, which is not always true. For example, bishops are more powerful (more valuable) in the endgame rather than at the beginning, so many programs incorporate non-linear combinations of the features and weights.

Figure 3 demonstrates the overall performance of a game-playing program. The more knowledgeable the evaluation function is, the stronger the program is. Similarly, the faster the search is (the deeper the game tree could be searched), the stronger the program is. However, typically, the more complicated an evaluation function is, the slower it becomes. Therefore, the right balance between the quality of the evaluation function and its speed is crucial for a successful game-playing program.

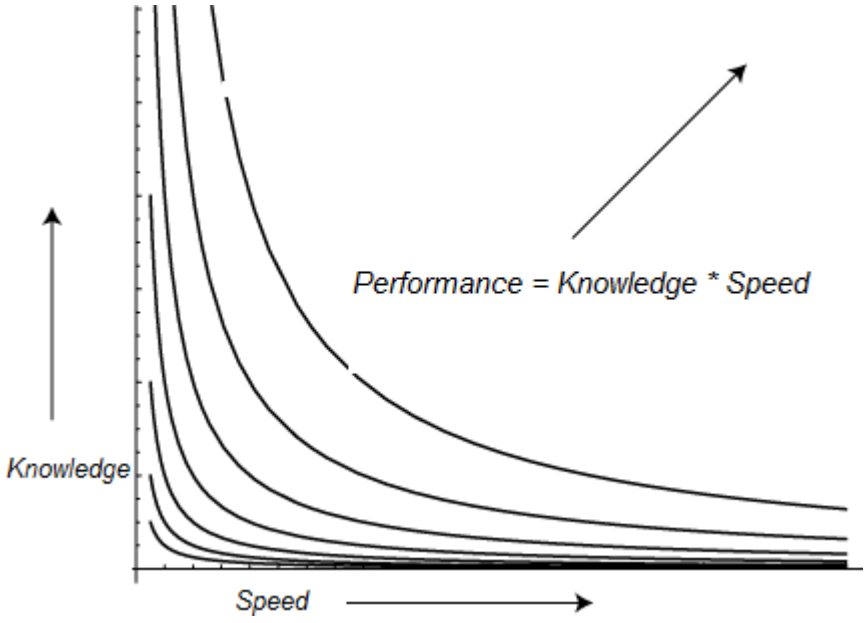


Figure 3. Game playing program performance: knowledge vs. speed [54]

2.9 Quiescence Search

The most straightforward approach for limiting the search space for the Minimax and Alpha-Beta algorithms is by setting a fixed depth limit so that the $Cutoff\_Test(state, depth)$ is positive for all depths greater than some fixed valued d. It also returns true for all terminal positions (leaves). Typically, depth d is chosen in a way that the search completes within an allowed time slot. However, the more preferable approach would be iterative deepening, which allows a flexible search process – the program returns the move selected by the deepest completed search. Another advantage is that iterative deepening provides helpful estimations on move ordering.

However, this simple approach can lead to errors. For example in chess, the current evaluation of a given position could be incorrect if the opponent can capture a figure in the next move. Thus, a more sophisticated cut-off test is needed. The evaluation function should be applied only to *quiescent* positions, which are those that are stable and unlikely to change value in the nearest game extension, for example, a chess position in which figure captures are possible is not considered quiescent for an evaluation function that considers material advantages. The non-quiescent position could be expanded further until a quiescence position is reached and the evaluation function could be applied. This extra search is called a *quiescence search* [26]. Mostly, it is restricted only to certain types of moves, such as captures and thus, will quickly resolve any uncertainties in the position evaluations.

Generally quiescence search is most important in chess; though this technique is successfully used in many different games.

2.10 Horizon Effect

The *horizon effect* is much more difficult to eliminate – Figure 4. It arises in situations when the program is facing considerable losses on the opponent's move (for example queen capture) and this impact is inevitable but can be avoided temporarily by use of delaying tactics. In this case, the program ignores this damage by pushing it out of its search range, which as a result could cause even more damage.

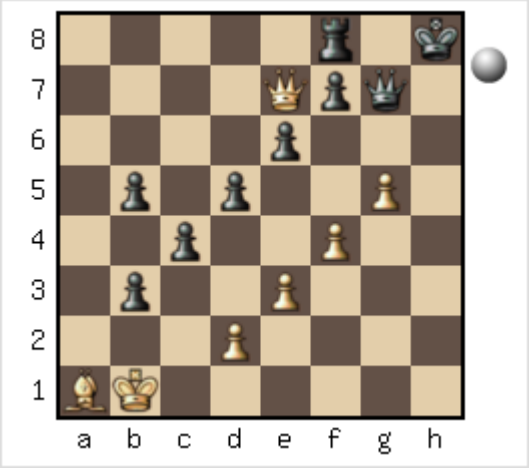


Figure 4. Horizon Effect [55]

One strategy to mitigate the horizon effect is the *singular extension*, introduced by Anantharaman et al. [34]. The main idea is to extend the search at cut-off nodes, if one move seems much better than all of the alternatives. This singular move is remembered and when the search depth limit is reached, the algorithm checks whether this singular extension is a legal move. If it is, the program allows the move to be considered in an additional search. This makes the search tree deeper, although these extensions are few and they do not require much extra time for the analysis.

2.11 Forward Pruning

So far, we have discussed search optimisation techniques that do not affect the correctness of the result. However, it is possible to do *forward pruning*, meaning that some moves can be pruned without performing a search based on specific criteria. Obviously, most human chess players consider only a few moves in each position while analysing the given situation.

One approach to implement forward pruning is the beam search: only a small number of the best moves is considered at each step for a deeper search, according to the evaluation function instead of checking all possible positions. Unfortunately, this approach is rather risky because there is no guarantee that the best move will not be pruned in this way.

The ProbCut, or probabilistic cut, is a selective search enhancement of the Alpha-Beta algorithm created by Michael Buro in 1994 and introduced in his Ph.D. thesis

[45]. The Alpha-Beta search prunes those nodes that are provably outside (a,b)-window, while ProbCut prunes nodes that are probably outside the window. As a result, the searched tree becomes narrower, allowing a deeper search. ProbCut and its improved variant Multi-ProbCut have proved effective in the game of Othello. Buro implemented this technique in his program Logistello and found that the ProbCut version matches the results found by a regular version in 64% of positions searched, even when the regular version was given an increased time limit [37].

2.12 Lookup Tables

There is no reason to perform a deep search in the beginning of a chess game just to conclude that the first move will be to move a pawn to e4. Books describing strategies of play in the opening and endgame in chess have been available for centuries. Therefore, many game-playing programs use *table lookup* instead of search for the first few moves for the opening and ending of the game.

For the opening, the computer mostly relies on human's expertise. The moves for each opening position, which are considered best by human players, are entered into the lookup table. Additionally, the computer can gather statistics from a database of previously played games in order to estimate the winning probability of each move. When the game starts, there are only a few choices and the program can rely on the lookup table. As the game proceeds and the program faces a position that it has not seen before, it should switch from the table lookup to search.

Towards the end of the game there are again fewer possible positions and thus we can store them in the lookup table. Computers are perfect in endgame search – their abilities go far beyond that which the human player can reach. Endgame databases are built in the following way. First, all legal positions with three pieces (figures in chess) are scanned. Some of them are checkmates, so they are marked in a table accordingly. Then, a retrograde Minimax search is performed: an "unmove" instead of the standard move is done. Any move that leads to the winning position is marked as a win. Continuing this way, we can build up a database for four pieces, five pieces, etc.

Using this technique, Ken Thomson [50] and Lewis Stiller [32] solved all chess endgames for up to five pieces and some with six pieces, making them available on the Internet. Thomson discovered [51] one case where a forced mate existed but required

262 moves – see Figure 5. This caused confusion in chess society because the rules of chess require a capture or pawn move to occur within every 50 moves.

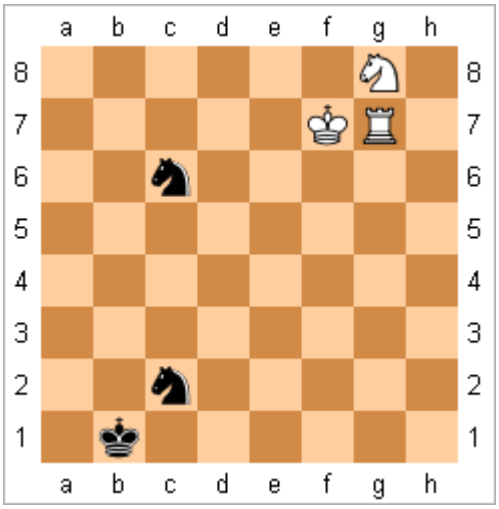


Figure 5. White mates in 262 moves [56]

Later work of Bourzutschky and Konoval solved all pawn-less six-piece positions and some seven-piece endgames and found an endgame position that requires 517 moves until the capture, which leads to a mate [33].

2.13 Stochastic Games

In real life, players often have to act in unpredictable and unforeseen situations. Typically, in games, dice rolls represent this randomness. This type of game is called a *stochastic game* or a game with the element of chance. Backgammon is a typical example of a game that combines both luck and skill. In this game, before each player' turn, dice are rolled and a list of legal moves is determined based on the result.

To represent the element of uncertainty, special chance nodes should be introduced to the game tree – see Figure 6. The branches leading from each chance node denote the possible dice rolls. Each branch is labelled with the roll and its probability. There are 36 ways to roll two dice, each equally likely; however, because 3-4 is the same as 4-3, there are only 21 unique outcomes. The six doubles (1-1, 2-2, 3-3, 4-4, 5-5 and 6-6) each have a probability of 1/36. The other 15 distinct rolls each have a probability of 1/18.

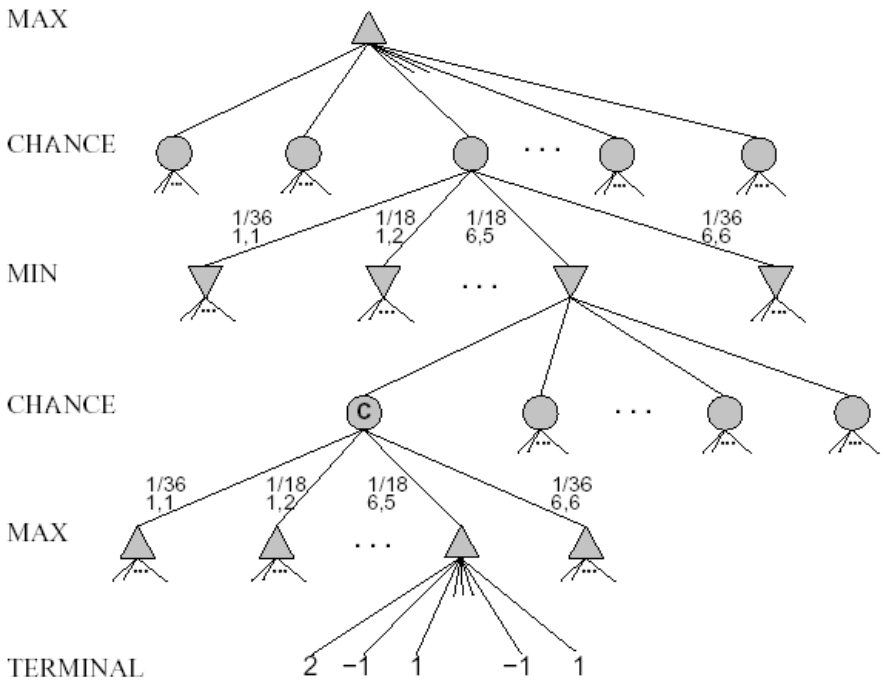


Figure 6. Stochastic Game Tree [11]

The next step is to understand the process of making correct decisions. We still need to find the move that leads to the best position. However, there is no defined Minimax value for this tree, so we can only operate with the concept of *expected* value – the average value of all possible outcomes over the chance nodes.

This leads to an *expectiminimax* value for games with the element of chance. The expectiminimax algorithm is a variant of the minimax algorithm and was first proposed by Donald Michie [52]. Terminal nodes and MIN / MAX nodes work in exactly the same way as before. For chance nodes, the expected value is computed based on all possible outcomes multiplied by the weighted probability of each action:

$$Expectiminimax(s) =$$

$$\begin{cases} Utility(s) & Terminal\_Test(s) \\ \max a \in Actions(s)\ Expectiminimax(Result(s,a)) & if\ Player(s) = MAX \\ \min a \in Actions(s)\ Expectiminimax(Result(s,a)) & if\ Player(s) = MIN \\ \sum P(r) * Expectiminimax\big(Result(s,r)\big) & Player(s) = CHANCE \end{cases} \quad (7)$$

where r represents all chance events (dice rolls) and $Result(s,r)$ is the same state "s" with the additional fact that the result of the dice rolls is "r".

2.14 Evaluation Functions for Games of Chance

Evaluation functions for stochastic games are similar to those in deterministic games but the presence of chance nodes introduces additional obstacles that must be taken into account. We might expect the same properties as in chess – the better the position is, the higher the estimation by valuation function should be. However, generally it is true that we should be more careful when considering this type of game.

Referring to Figure 7, we see that for a deterministic game, if leaf nodes are assigned with values [1, 2, 3, 4], then the best move is a2. With new values [1, 20, 20, 400] the situation is not changed; the best move is still a2. The exact values for the evaluation function don't affect the relative choice of the moves.

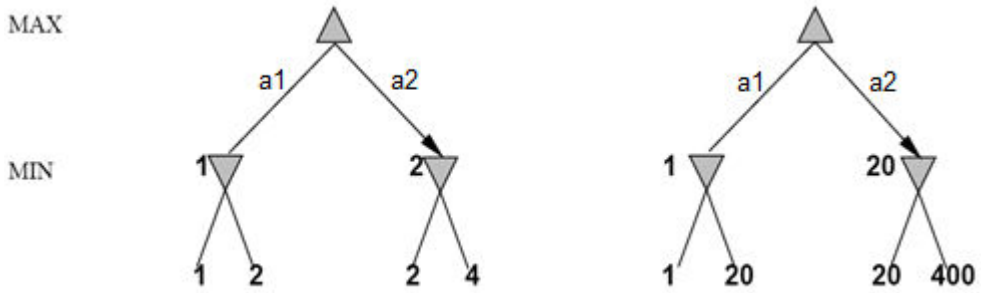


Figure 7. Monotonic transformation of evaluation function [11]

However, with non-determinism, the exact values do matter – see Figure 8. If we assign leaf values [1, 2, 3, 4] for a stochastic tree, then the best move is a1 but with values [1, 20, 30, 400] the situation changes completely; now a2 becomes the best move. To avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning in a given position. This is an important and general property of situations involving uncertainty.

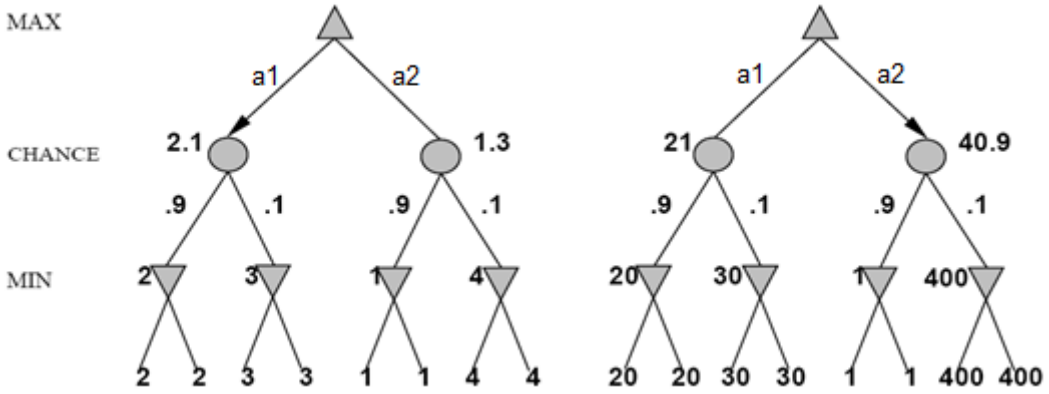


Figure 8. Positive linear transformation of evaluation function [11]

To solve a game tree with an element of chance it will take

$$O(w^{\mathrm{d}}n^{\mathrm{d}}) \qquad (8)$$

where w is width, d is depth and n is the number of distinct rolls. Even if the search depth is limited to some small depth, the extra cost compared with the Minimax algorithm makes it unrealistic to consider very far looking ahead in most games of chance. In backgammon, n constitutes 21, w is typically around 20 and so three plies is probably all we could manage.

However, some optimisation to the expectiminimax algorithm exists, such as Star1 and Star2 but their efficiency is still far from the improvement that Alpha-Beta makes to Minimax [35].

2.15 Imperfect Information Games

We have discussed *perfect information* games. These games describe the situation in which each player has full information available in order to determine possible moves (or combinations of legal moves). For example, in chess each player can see all the pieces on the board all the time.

However, in many situations only limited amounts of information or certain details are available. A typical example is card games where player's cards are hidden from the opponent. These types of games are called *imperfect information* games. As the

name suggests, these games are qualitatively different from those games described before.

2.16 Kriegspiel (chess)

Kriegspiel (*war game* - German) is a chess variant invented by Michael Temple in 1899 and based upon the original Kriegspiel developed by Georg von Rassewitz in 1812. In this game, each player can see his own pieces but not the opponent's. For this reason, this game requires a referee, third person or computer that can observe and control the whole position during the game. During his turn, a player makes an attempt of a move. If the referee declares the move illegal, the player tries again. If it is legal, then the move stands. Each player is given the information about checks and figure captures. They may also ask the referee if there are any legal captures with a pawn. Because the position of the opponent's pieces is unknown, Kriegspiel is a game with imperfect information – Figure 9. Chess Kriegspiel derives from a war game that was used in 19th century in Germany to train military officers [17].

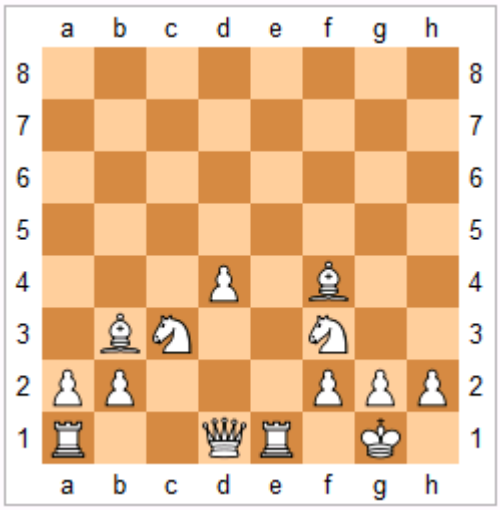


Figure 9. Kriegspiel, the game in progress. Position as seen by White player [17]

Initially, Kriegspiel may seem impossible but humans manage it quite well and computer programs are beginning to catch up. The representation of the game uses the concept of the *belief state* – the set of all logically possible board positions given a complete historical track of moves to date. In the initial position, White's belief state is

singleton, as Black has not yet moved. After the first move and Black's reply, White's belief state contains 20 positions, as there are a possible 20 replies. Keeping track of the belief state while the game progresses is the problem of *state estimation* – how likely is the given position to occur after a series of moves.

The concept of imperfect information substantially changes the game strategy. Each player's goal is not just to move figures to the best squares but also to minimise the information that the opponent has about their location. Hence, players are required to play somewhat randomly.

These considerations lead to the theoretical notion of an *equilibrium* solution, which specifies an optimal randomised strategy for each player. Computing equilibrium is a hard problem even for small games and out of question for the Kriegspiel. At present, the design of effective algorithms for general Kriegspiel play is an open research topic.

2.17 Card Games

Card games provide examples of stochastic imperfect information games, where the missing information is generated randomly in the beginning of the game. For example, in many games, cards are dealt at the initial step and these cards are not visible by the opponent. Such games include poker, bridge, whist and some others.

At first sight, it might seem that card games are similar to the games with the element of chance because cards are dealt randomly and define each player's legal moves. Although this analogy is incorrect, we can assume that all "dice" are rolled in the beginning. This leads us to a simple and effective algorithm: all possible deals of invisible cards are considered and solved individually as for a perfect information game. Then, the move that has the best expected outcome over all possible deals is chosen. Suppose that each deal s occurs with probability P(s). Then we are looking for a move

$$argmax\ P(s)\ Minimax(Result(s,a)) \tag{9}$$

Here, if we are able to search the whole tree, we run Minimax, otherwise we run $H\_Minimax$.

The main problem is that in real games the number of possible combinations is quite large. For example, in bridge play there are 10 million deals. Solving even one deal is difficult, so solving the whole game is out of question. That is why a Monte Carlo simulation is widely used. Instead of analysing all deals, we consider only N randomly chosen samples. The probability of deal "a" appearing in the sample is proportional to P(s):

$$argmax\ \frac{1}{N}\sum_{i=0}^{N} Minimax(Result(s_i, a)) \quad (10)$$

The described method provides a very good approximation even for small N, say 100 to 1000. However, it has significant limitations because it does not consider belief states as it assumes that every future state will be one of perfect knowledge. The approach never selects actions that gather information from the opponent, nor will it choose actions that hide information from the opponent. As a result, it will never bluff in poker because it assumes that the opponent can see its cards. To overcome these issues, more sophisticated algorithms are required [11].

2.18 Game Classification

According to game type and its properties, we could build a simple table with a game grouping or classification (Table 1).

Table 1. Finite zero-sum games

	deterministic	*chance*
perfect information	chess, checkers, go, othello	backgammon, monopoly, roulette
imperfect information	battleships, kriegspiel, rock-paper-scissors	bridge, poker, scrabble

The most widespread games are deterministic perfect information games, such as chess, checkers, go, etc. This group is considered the “simplest” games. The next two groups are perfect information games with the element of chance (including backgammon, monopoly, etc.) and deterministic imperfect information games. These are more complex due to the number of positions searched or states required for analysis. The last group is considered the most sophisticated – imperfect information games with the element of chance.

2.19 Conclusion

In this chapter we gave an introduction to game theory aspects. We also described challenging problems and typical solutions to cover various areas of artificial intelligence. In order to make our overview complete we included brief analysis of different types of games, though our further research will focus on the first group – deterministic perfect information games.

3 SEARCH ALGORITHMS

In this section, we describe state-of-the-art game tree search algorithms and see their evolution. Each algorithm is followed by a description and pseudo-code for deeper understanding. These implementations have been used in our experiments.

3.1 Minimax

Minimax is a base algorithm for determining the score in a zero-sum game according to the evaluation function. For simplicity, MAX and MIN players have their own search routines [18].

```
int maxi( int depth ) {
    if ( depth == 0 ) return evaluate();
    int max = -oo;
    for ( all moves) {
        score = mini( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}

int mini( int depth ) {
    if ( depth == 0 ) return -evaluate();
    int min = +oo;
    for ( all moves) {
        score = maxi( depth - 1 );
        if( score < min )
            min = score;
    }
    return min;
}
```

Figure 10. Minimax algorithm

3.2 Negamax

Negamax is a commonly used standard implementation of Minimax but instead of having two subroutines for each player, negated score is used [19].

```
int negaMax( int depth ) {
    if ( depth == 0 ) return evaluate();
```

```
    int max = -oo;
    for ( all moves)  {
        score = -negaMax( depth - 1 );
        if( score > max )
            max = score;
    }
    return max;
}
```

Figure 11. Negamax algorithm

3.3 Alpha-Beta

The Alpha-Beta search algorithm (Alpha-Beta pruning or Alpha-Beta Heuristic) is a significant improvement on the Minimax algorithm, which eliminates large portions of the search tree whilst being able to compute the correct Minimax value of a game tree.

The algorithm maintains two values: alpha and beta (called the alpha-beta window). They represent the minimum score that the MAX player is guaranteed to obtain and maximum score that the MIN player is guaranteed to receive, respectively [20].

```
int alphaBeta( int alpha, int beta, int depthleft ) {
   if( depthleft == 0 ) return quiesce( alpha, beta );
   for ( all moves)  {
      score = -alphaBeta( -beta, -alpha, depthleft - 1 );
      if( score >= beta )
         return beta;   // beta-cutoff
      if( score > alpha )
         alpha = score; // alpha acts like max in MiniMax
   }
   return alpha;
}
```

Figure 12. Alpha-Beta algorithm

3.4 Null / Zero Window

The Alpha-Beta algorithm has varying parameters – alpha-beta window. On average, the smaller the window, the faster the search could be performed. Therefore, another way of reducing the search space is by using a search window that is as small as possible. The null window or zero window approach, which is naturally a boolean test, checks whether a move produces a better or worse result than a passed value. Many algorithms incorporate this idea [11].

3.5 Principal Variation Search (PVS)

PVS (Principal Variation Search) is an enhancement of Alpha-Beta, based on null or zero window searches of none-PV-nodes to prove whether a move is worse or not than the already safe score from the principal variation [1, 10, 21].

```
int pvSearch( int alpha, int beta, int depth ) {
   if( depth == 0 ) return quiesce( alpha, beta );
   bool bSearchPv = true;
   for ( all moves)  {
      make
      if ( bSearchPv ) {
         score = -pvSearch(-beta, -alpha, depth - 1);
      } else {
         score = -pvSearch(-alpha-1, -alpha, depth - 1);
         if ( score > alpha ) // in fail-soft ... && score < beta )
is common
            score = -pvSearch(-beta, -alpha, depth - 1); // re-
search
      }
      unmake
      if( score >= beta )
         return beta;   // fail-hard beta-cutoff
      if( score > alpha ) {
         alpha = score; // alpha acts like max in MiniMax
         bSearchPv = false;  // *1)
      }
   }
   return alpha; // fail-hard
}
```

Figure 13. PVS algorithm

3.6 NegaScout

NegaScout is an Alpha-Beta enhancement. The improvements rely on a Negamax framework and some fail-soft issues concerning the two last plies, which do not require any re-searches [3, 4]. Search-wise PVS and NegaScout are similar except for deep cut-offs [22].

```
int NegaScout ( position p; int alpha, beta )
{                      /* compute minimax value of position p */
   int b, t, i;
   if ( d == maxdepth )
      return quiesce(p, alpha, beta);           /* leaf node */
   determine successors p_1,...,p_w of p;
   b = beta;
   for ( i = 1; i <= w; i++ ) {
```

```
      t = -NegaScout ( p_i, -b, -alpha );
      if ( (t > a) && (t < beta) && (i > 1) )
         t = -NegaScout ( p_i, -beta, -alpha ); /* re-search */
      alpha = max( alpha, t );
      if ( alpha >= beta )
         return alpha;                              /* cut-off */
      b = alpha + 1;                  /* set new null window */
   }
   return alpha;
}
```

Figure 14. NegaScout algorithm

3.7 NegaC*

*NegaC** incorporates an idea to turn a Depth-First into a Best-First search like MTD(f) to utilise null window searches of a fail-soft Alpha-Beta routine and use the bounds that are returned in a bisection scheme [5, 23].

```
int negaCStar (int min, int max, int depth) {
   int score = min;
   while (min != max) {
      alpha = (min + max) / 2;
      score = failSoftAlphaBeta (alpha, alpha + 1, depth);
      if ( score > alpha)
         min = score;
      else
         max = score;
   }
   return score;
}
```

Figure 15. NegaC algorithm

3.8 SSS* / Dual*

*SSS** and its counterpart *Dual** are search algorithms that conduct a state space search traversing a game tree in the best-first fashion, similar to that of the A* search algorithm and retain global information about the search space. They search fewer nodes than Alpha-Beta in fixed-depth Minimax tree search [2, 24].

```
int MT-SSS*( n )
{
   g := +oo;
   do {
      G := g;
      g := Alpha-Beta(n, G-1, G );
```

```
    } while (g != G);
    return g;
}
```

Figure 16. SSS* algorithm

```
int MT-DUAL*(n)
{
    g := -oo;
    do {
        G := g;
        g := Alpha-Beta(n, G, G+1 );
    } while (g != G);
    return g;
}
```

Figure 17. Dual* algorithm

3.9 MTD(f)

MTD(f), the shortened name for MTD(n, f), which stands for Memory-enhanced Test Driver with node n and value f. MTD is the name of a group of driver-algorithms that search Minimax trees using null window Alpha-Beta with transposition table calls. In order to work, MTD(f) needs a first guess as to where the Minimax value will turn out to be. The better that first guess is, the more efficient the algorithm will be on average, because the better it is, the fewer passes the repeat-until loop will have to do to converge on the Minimax value [6-9, 25].

```
int mtdf(int f, int depth) {
    int bound[2] = {-oo, +oo}; // lower, upper
    do {
        beta = f + (f == bound[0]);
        f = alphaBetaWithMemory(beta - 1, beta, depth);
        bound[f < beta] = f;
    } while (bound[0] < bound[1]);
    return f;
}
```

Figure 18. MTD(f) algorithm

3.10 Conclusion

In this chapter we gave an overview of existing search algorithms. All these algorithms were implemented and thoroughly analysed in our research. We will also

use them as a reference for performance comparison and further improvements of algorithms.

4 SELECTIVE SEARCH

In this section, we describe alternative approaches to game tree search algorithms, which are based on selective searches. Then we give descriptions for each algorithm, followed by pseudo-code for deeper understanding.

4.1 Probabilistic Forward Pruning

Game tree search remains one of the challenging problems in AI and an area of active research. While classical results have been achieved based on Alpha-Beta pruning, there are a lot of further enhancements and improvements, including NegaScout, NegaC*, PVS, SSS* / Dual*, MTD(f) and others, as we have discussed in the previous chapter.

Alternative approaches also exist. For instance, Berliner's algorithm B*, which uses interval bounds, allows the selection of the optimal move without the need to compute the exact game value [36].

While these algorithms are based on optimal search (always returning an optimal solution), recent approaches incorporate approximation ideas. Mostly, they are based on probabilistic forward pruning techniques where we are trying to prune less optimistic nodes and sub-trees based on heuristics or a shallow search.

4.2 Approximation and near Optimal Search

Recently, approximation search paradigms have become popular. This may be due to the following reasons. Firstly, classical search algorithms, which try to find optimal solutions, are close to their theoretical limits and it is difficult to improve them further. Secondly, these algorithms rely on the quality of the evaluation functions, which are not optimal by their nature – they are an approximation of the utility value of board position of which we are not aware. Therefore, if our search is based on approximate position estimations, then there is no reason why we cannot change the nature of the tree search and also make it approximate.

In fact, it is possible and a lot of research has been done in this direction. We will examine the most important algorithms and describe them in detail.

4.3 ProbCut

ProbCut is a selective search modification of Alpha-Beta. It assumes that estimation of a node could be done based on a shallow search. It excludes sub-trees from the search, which are probably irrelevant to the main line of play. This approach has been successfully used in the Othello program Logistello, significantly improving the playing strength [37].

```
int AlphaBeta(int height, int alpha, int beta)
{
    int i, max, val;
    POSDELTA delta;

    if (Leaf(&pos, height))              /*     leaf-position? */
        return Eval(&pos);               /* yes => evaluate it */

    /* location of the selective extension */

    max = alpha;                         /* initialize maximum */
    for (i = 0; i < pos.movenum; i++) {  /*   forall moves ... */
        Move(&pos, pos.move[i], &delta); /* make move and save */
                                         /*   changes in delta */
        val = -AlphaBeta(height - 1, -beta, -max);  /* negamax */
        Undo(&pos, &delta);                 /* restore old pos */
        if (val > max) {
            if (val >= beta) return val;        /*      cutoff */
            max = val;                          /* new maximum */
        }
    }

    return max;
}
```

Figure 19. A negamax implementation of the alpha-beta algorithm

Implementation of the ProbCut extension is shown in Figure 19.

```
#define PERCENTILE 1.5              /* i.e. p ca. 0.93         */
#define DP         4                /* depth of shallow search */
#define D          8                /* check height            */

int AlphaBeta(int height, int alpha, int beta)
{
    ...
    if (height == D) {
        int bound;

        /* v >= beta with prob. of at least p? yes => cutoff */

        bound = round((+PERCENTILE*sigma + beta - b)/a);
```

```
        if (AlphaBeta(DP, bound - 1, bound) >= bound) return beta;

        /* v <= alpha with prob. of at least p? yes => cutoff */

        bound = round((-PERCENTILE*sigma + alpha - b)/a);
        if (AlphaBeta(DP, bound, bound + 1) <= bound) return alpha;
    }
    ...
}
```

Figure 20. The ProbCut extension

4.4 Multi-ProbCut

Multi-ProbCut (MPC) is a further improvement of ProbCut that uses additional parameters and pruning thresholds for different stages of the game. This new approach has improved the strong Othello program Logistello, considerably.

Multi-ProbCut generalises the selective search procedure ProbCut. When thinking about the weaknesses of ProbCut and its simplifying assumptions, MPC introduces several potential improvements. It allows forward cuts at various heights after shallow searches of increasing depths. Combined with the new evaluation function, the resulting playing strength increase is equivalent to a speed-up factor of more than ten [38].

```
const int MAX_STAGE  = 64; // e.g. disc number
const int MAX_HEIGHT = 13; // max. check height
const int NUM_TRY    = 2;  // max. number of checks

// ProbCut parameter sets for each stage and height
struct Param {
    int   d;       // check depth
    float t;       // cut threshold
    float a, b, s; // slope, offset, std.-dev.
} param[MAX_STAGE+1][MAX_HEIGHT+1][NUM_TRY];

Position pos;

int MPC(int height, int alpha, int beta) {
    int i, max, val;
    PosDelta delta; // contains undo information

    if (height == 0) return pos.eval(); // leaf

    // check part:
    if (height <= MAX_HEIGHT) {
        for (i = 0; i < NUM_TRY; i++) {
            int bound;
            Param &pa = param[pos.stage][height][i];
            if (pa.d < 0) break; // end-marker reached?
```

```
            // is v_height >= beta likely?
            bound = round((pa.t*pa.s + beta - pa.b)/pa.a);
            if (AlphaBeta(pa.d, bound - 1, bound) >= bound)
                return beta; // yes => cutoff

            // is v_height <= alpha likely?
            bound = round((-pa.t*pa.s + alpha - pa.b)/pa.a);
            if (AlphaBeta(pa.d, bound, bound + 1) <= bound)
                return alpha; // yes => cutoff
        }
    }

    // the remainder of the alpha-beta algorithm:
    max = alpha;
    for (i = 0; i < pos.move_num; i++) {
        pos.make_move(i, delta);
        val = -MPC(height - 1, -beta, -max);
        pos.undo_move(delta);
        if (val > max) {
            if (val >= beta) return val;
            max = val;
        }
    }
    return max;
}
```

Figure 21. A C implementation of Negamax MPC

Further experiments demonstrated that this approach could be efficiently used in chess [39].

4.5 Multi-cut

Multi-cut is a speculative pruning technique that takes into account, not only the probability of cutting off relevant lines of play but also the chances of such a wrong decision affecting move selection at the root of the search tree [40].

Expected cut-nodes, where many moves have a good potential of causing Alpha-Beta cut-off, are less likely to become all-nodes and consequently, such lines are unlikely to become part of a new principal variation. This observation forms the basis for forward-pruning schemes called Multi-cut Alpha-Beta pruning [40].

```
// M is the number of moves to look at when checking for mc-prune.
// C is the number of cutoffs to cause an mc-prune, C < M.
// R is the search depth reduction for mc-prune searches.

int zwSearch( int beta, int depth, bool cut) {
    if ( depth <= 0 ) return quiesce( beta-1, beta );
```

```
    if ( depth >= R && cut ) {
        int c = 0;
        for ( first M moves )
            score = -zwSearch( 1-beta, depth-1-R, !cut);
            if ( score >= beta ) {
                if ( ++c == C )
                    return beta; // mc-prune
            }
        }
    }
    for ( all moves ) {
        score = -zwSearch( 1-beta, depth-1, !cut);
        if ( score  >= beta )
            return beta;
    }
    return beta - 1;
}
```

Figure 22. A C implementation of a Multi-Cut zero window search

Multi-Cut inside is a null window or zero window search of a fail-hard PVS framework, applied at expected cut-nodes.

4.6 RankCut

RankCut is a domain-independent forward pruning technique that exploits move ordering and prunes once it is determined that no better move is likely to be available. It has been implemented in an open source chess program, CRAFTY, where RankCut reduced the game-tree size by approximately 10% to 40% for search depths 8–12, whilst retaining tactical reliability [41].

```
RankCut(alpha, beta, depth)
    if depth == 0 then
        return LeafEvaluate()
    pruneRest ← false
    for move ← NextMove() do
        r ← 0
        Compute(f_i)
        if (pruneRest || ∏(f_i) < t) then
            r ← DepthReduction()
            pruneRest ← true
        score ← -RankCut(-alpha, -beta, depth - 1 - r)
        if alpha < score then
            pruneRest ← false
            alpha ← score
        if score ≥ beta then
            return score
    return score
```

Figure 23. RankCut pseudocode

4.7 Game Tree Search with Adaptive Resolution

Game Tree Search with Adaptive Resolution is an approach where the value returned by the modified algorithm, called Negascout-with-resolution, differs from that of the original version by at most R. Experiment results demonstrate that Negascout-with-resolution yields a significant performance improvement over the original algorithm on the domains of random trees and real game trees in Chinese chess [42].

```
Negascout-with-resolution(position p, value alpha, value beta,
integer depth, integer R)

Input: position p, value alpha, value beta, integer depth, integer R
Output: value m

if depth = 0 then
  return E(p)
end
m:= −∞;
n:= beta;
foreach p_i ∈ B(p) do
  t:= -Negascout-with-resolution(p_i,-n,-max{alpha,m}, depth − 1,R);
  if ⌊t/R⌋ > ⌊m/R⌋ then /* apply the resolution scheme to compare the
  values. */
    if n = beta then
      m := t;
    else
      m := -Negascout-with-resolution(p_i,−beta,-t, depth − 1, R);
    end
  end
  if ⌊m/R⌋ ≥ ⌊beta/R⌋ then /* apply the resolution scheme when
  deciding a possible beta cut. */
    return m;
  end
  n := max{alpha,m} + 1;
end
return m;
```

Figure 24. Negascout-with-resolution pseudocode

4.8 Conclusion

In this chapter we gave an overview of existing algorithms which are based on selective search. We use these ideas in our further research and show how proposed algorithm (Chapter 7) differs from current approaches.

PART II

RESEARCH WORK AND RESULTS

5 FUZZIFIED TREE SEARCH

In this section, we will describe a new approach for game tree searches and will then look into the structure and implementation details of the algorithm.

5.1 The Fuzzy Approach

We propose a new approach based on an attempt to implement a human way of thinking adapted to logical games. A human player rarely, or almost never, evaluates a given position precisely. In many cases, the selection process is limited to rejecting less promising nodes and making certain that the selected option is better than others. An important point is that we are not interested in the exact position evaluation but rather in the node, which guarantees the highest outcome.

Here, we explain the given problem in detail.

We could look at our game tree from a relative perspective, for example, “whether this move is better or worse than some value X” (Figure 25). At each level, we identify whether a sub-tree satisfies some “greater or equal” criteria. Therefore, executing the search algorithm, for instance, with argument 5, we obtain the information that the left branch has a value less than 5 and the right branch has a value greater than, or equal to 5. We do not know the exact sub-tree evaluation but we have found the move, which leads to the best result.

In this case, different cut-offs are possible:

- at max level, if the evaluation is greater (or equal) than the search value;
- at min level, if the evaluation is less than the search value.

In the given example, the reduced nodes are shown with the dashed line. Comparing it with Figure 2, it can be seen that not only more cut-offs are possible but also, pruning occurs at a higher level resulting in a better performance.

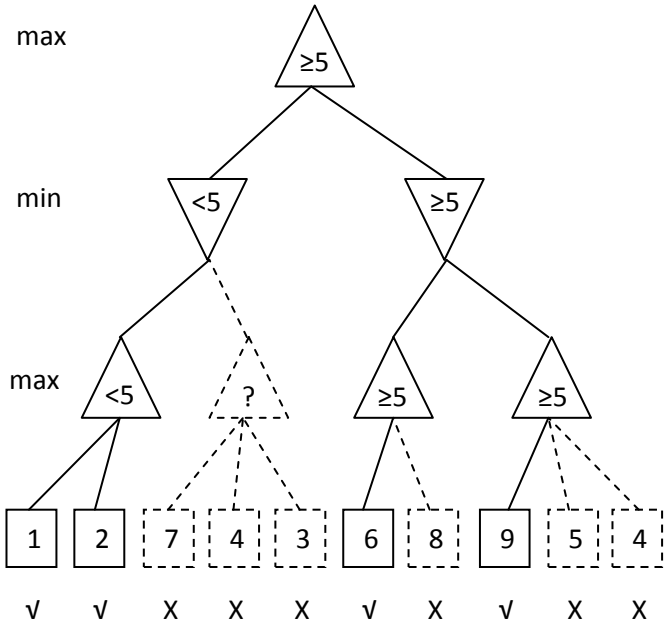


Figure 25. Fuzzy best node approach

In this approach, the best/worst cases are the same as for the Alpha-Beta pruning: $O(w^{d/2})$ for the best case, as only one branch should be checked at cut-off level and $O(w^{d})$ for the worst case, because all nodes should be checked (w is width and d is depth of the tree). However, in general, in the presented approach, cut-offs are more often possible.

If we use a geometric interpretation and put our sub-tree Minimax values on the coordinate axis, then our task is to separate/divide branches so that only one branch would have a value higher than the test value. Figure 26 illustrates our previous example. The alpha-beta window is initially set to leaf node range $\alpha = 0$, $\beta = 10$; then the following test values are used X_1, X_2 and X_3. If value X_2 is chosen, then the successful separation is obtained after the first iteration, as we know that the second sub-tree has a higher estimation. If values X_1 or X_3 are chosen, then no separation is possible at this point – both values are on the same side of the test value. In this case, the algorithm continues with the reduced alpha-beta search window: 1) $\alpha = X_1$ in the first case, or 2) $\beta = X_3$ in the second.

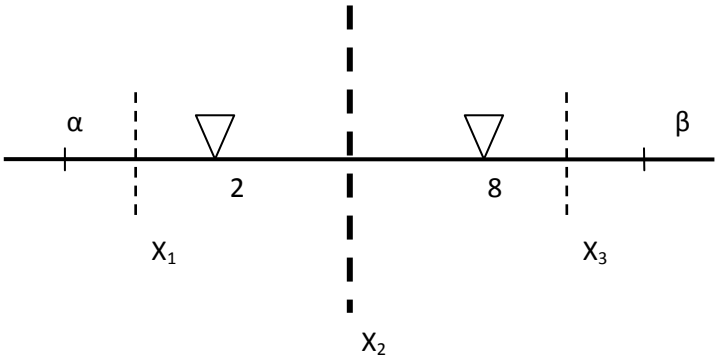


Figure 26. Geometric interpretation of separation in the Fuzzified game tree search

In a game tree with three or more sub-trees, the algorithm workflow remains the same. Our task is to separate sub-trees in a way that only one branch has a higher estimation than the test value. However, for a tree with three sub-trees more cases are possible – 0, 1, 2 and 3 branches fall on one side of the separation line. In this case, the alpha-beta window is reduced correspondingly and the algorithm proceeds with the next iteration.

Comparing this with existing algorithms, such as MTD(f), it needs a first guess at where the Minimax value will turn out to be in order to work. If you initially feed the Minimax value to MTD(f), it will only do two passes, which is the bare minimum: one to find an upper bound of value x and one to find a lower bound of the same value [27, 28].

In the presented algorithm, it is possible to find the best move after a single iteration and we are not limited to an accurate first guess. For the presented example, any value from interval 3...7 (inclusive) would apply.

5.2 The Fuzzified Search Algorithm

Best Node Search (BNS) is a new game tree search algorithm based on the idea described in the previous section. The main difference between the classical approach and the proposed algorithm is that BNS does not require knowledge of the exact game tree Minimax value in order to select a move. We only need to know which sub-tree has the higher estimation. By iteratively performing search attempts, the algorithm can obtain information about which branch has a higher estimation without knowing the exact value. Therefore, less information is required and as a result, the best move can

be found faster – the total number of searched nodes is smaller and the total algorithm execution time is reduced in comparison with the algorithms based on the exact game tree evaluation.

The presented algorithm uses a standard Alpha-Beta search with 'zero window'. The proposed implementation relies on the transposition tables but a variation without memory (transposition tables) usage is also possible. While scanning a game tree, the algorithm checks all sub-trees at root level and returns the node that leads to the best result. In general, BNS is expected to be more efficient compared with the classical algorithms in terms of the number of nodes checked, as it does not obtain additional information, which in many cases is not required – the exact game tree Minimax value.

The BNS algorithm is given in Figure 27, which makes use of the following functions:

1. NextGuess() – returns the next separation value to be tested by the algorithm;
2. AlphaBeta() – Alpha-Beta search with Zero Window (Null Window), which performs a boolean test on whether a move produces a score that is worse or better than the passed value.

All sub-trees are tested with the separation values (this information is stored in the transposition tables and reused in subsequent iterations). If one and only one branch exceeds the test value, then the best node is found. If all branches have smaller estimations, then the number of sub-trees that exceeds the separation test value remains the same and the beta value is reduced. If several nodes exceed the test value, then correspondingly `subtreeCount` is updated, the alpha value is updated to test value and the algorithm continues with the next iteration. If a single sub-tree that exceeds the test value cannot be found and the Alpha-Beta range is reduced to 1, this means that several sub-trees have the same estimation and we can choose any of them.

```
function BNS(node, α, β)
    subtreeCount := number of children of node
    do
        test := NextGuess(α, β, subtreeCount)
        betterCount := 0
        foreach child of node
            bestVal := -AlphaBeta(child, -test, -(test - 1))
            if bestVal ≥ test
                betterCount := betterCount + 1
                bestNode := child
        update number of sub-trees that exceeds separation test
value
        update alpha-beta range
    while not((β - α < 2) or (betterCount = 1))
    return bestNode
```

Figure 27. The BNS algorithm

One of the main parts of this algorithm is the method `NextGuess(α, β, subtreeCount)`, which returns the next value to be checked by the algorithm. In the simplest case, it could be a formula based on a linear distribution – the Alpha-Beta range is proportionally divided into sections according to the sub-tree count:

$$NextGuess = \alpha + (\beta - \alpha) * \frac{subtreeCount - 1}{subtreeCount} \quad (11)$$

where alpha and beta are the lower and the upper bounds of the search window, respectively; `subtreeCount` is the number of sub-trees that satisfies the previous test call (the branches that have higher estimations than the test value).

Moreover, dynamic adjusting of the separation value provided by the function NextGuess is also possible. If during our search, we can conclude that we should update the separation value in order to increase the probability of successful separation, then we can apply it. For example, if the current node has a higher or equal estimation than the test value, then it is reasonable to increase this test value and check the remaining nodes with the new updated test value.

However, the best algorithm performance is achieved after statistical training or analytical game tree evaluation, which gives accurate information about the resulting game tree value distributions. As a result, NextGuess also becomes more precise.

Some algorithms, such as MTD(f), benefit from an accurate "first guess" – at where the Minimax value will turn out to be. On average, the better the first guess is, the more efficient the algorithm will be.

The BNS algorithm can benefit significantly from a good separation value as well. On average, the better the separation value is, the faster the best node will be found. One approach to improve the performance of the algorithm is to enhance it through statistical training. The main idea is to collect statistical information over a series of games in order to analyse and find optimal separation values for the algorithm.

Another approach is based on the game tree analytical evaluation, where probability density functions and cumulative distribution functions are applied, in order to calculate the most probabilistic outcome at each level of the game tree.

5.3 BNS Enhancement through Statistical Training

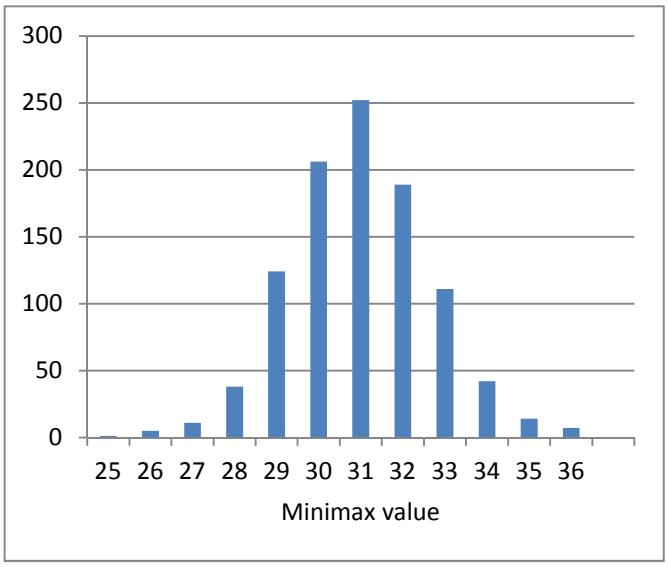
As we already mentioned, the BNS algorithm can benefit greatly from good a separation value. Thus, self-training becomes an important part of the BNS algorithm, because it helps us to tune separation test values used by algorithm during consecutive search attempts and results in reduced search space and improved performance.

In this section, we propose a new multi-dimensional statistical approach, which is developed to work in conjunction with the BNS algorithm.

It is possible to collect these statistics before the game starts by analysing multiple test data or online during the game process by reusing previous estimations.

Table 2. Game tree Minimax value distribution over 1000 trees

Minimax value	Tree count
25	1
26	5
27	11
28	38
29	124
30	206
31	252
32	189
33	111
34	42
35	14
36	7
	1000



The statistical approach for finding an initial value (first guess) can be demonstrated in the following example. One thousand game trees were generated with fixed structure and randomly assigned values for leaf nodes in a specific range (for the given example, the following values were used – width 2, depth 14, leaf node values are in the interval [0; 80]). For these game trees, statistics were collected and the results are shown in Table 2.

It can be seen that there are 252 trees with a Minimax value of 31 and there is only one tree out of one thousand with a Minimax value of 25. These statistics are used to determine the first guess in the MTD(f) algorithm and in all tests it was called with argument f = 31 showing the best results.

However, this information does not provide additional benefits and therefore, a new approach is proposed, i.e., single-dimension statistics is extended into two-dimension statistics, meaning the collection of information on all possible pairs – for each sub-tree in our binary tree. As a result, we have a matrix showing a number of trees having respectively one sub-tree value (columns) and other sub-tree value (rows) (Table 3). Because of reasons of symmetry (according to the main diagonal), only one half is shown. The tree count column has summed up matrix values in the row resulting in the previous single-dimension statistics (Table 2).

It can be seen that there are 78 trees that have corresponding sub-trees with branch values of 31 and 29 (in this case, the tree Minimax value is 31).

Table 3. Two dimensional game sub-tree distribution over 1000 trees

	23	24	25	26	27	28	29	30	31	32	33	34	35	36	Tree count
23	0														0
24	0	0													0
25	0	1	0												1
26	0	0	2	3											5
27	0	0	5	3	3										11
28	0	1	0	12	12	13									38
29	0	0	2	10	35	43	34								124
30	1	2	6	9	26	58	71	33							206
31	0	0	6	10	27	41	78	57	33						252
32	0	1	3	13	17	30	32	41	38	14					189
33	0	0	1	2	8	12	26	28	21	11	2				111
34	0	0	0	1	3	5	13	8	6	2	2	2			42
35	0	0	0	0	0	2	4	3	2	3	0	0	0		14
36	0	0	0	0	0	0	1	2	2	1	1	0	0	0	7
															1000

The BNS algorithm divides the search interval into parts and verifies whether sub-tree values stay in different parts or not. If one branch value is less than the separation value and another branch value is higher, then the algorithm immediately returns a better move and stops its work. If the branch values lie in the same part, then the interval is reduced and the algorithm continues with an updated Alpha-Beta window. Thus, the algorithm becomes more efficient with an accurate first guess, when most of the game trees get separated into parts after the first iteration.

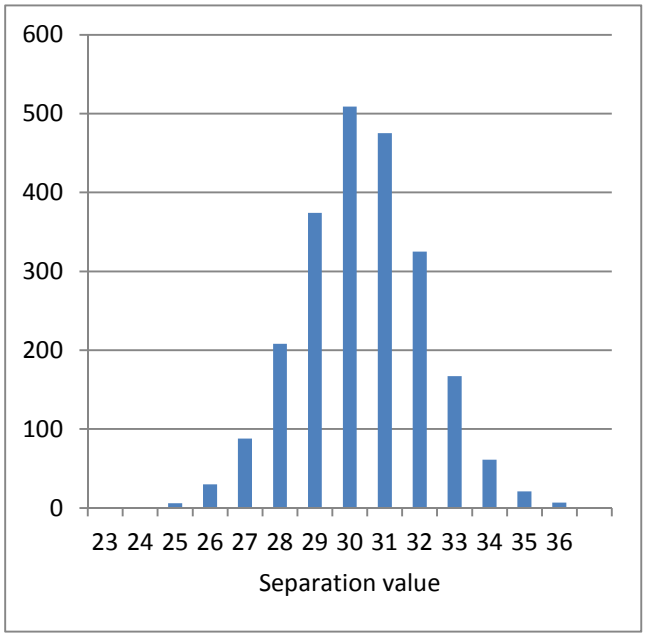
The grayed-out rectangle in Table 3 gives us the separation distribution for X = 30. All marked cells represent trees with one branch greater or equal than 30 (by row) and the other branch less than 30 (by column). It means that all these trees will be separated into parts after the first iteration. To calculate the number of trees for the separation value X = 30, we need to sum up all the marked cells. For the given example, 509 trees will become separated.

Therefore, to find the value of X when the highest number of trees will be divided, we need to build the remaining rectangles along the main diagonal for each X value and sum up the cells bounded by X along the axis, as was done in the previous example. The resulting table is shown in Table 4.

As can be seen from Table 4, the best results are given with X = 30, meaning that if we call the BNS algorithm with argument 30, then 509 game trees will be divided into two parts and the best node will already be found after the first iteration. Thus, a trained BNS algorithm is more efficient and if we develop this idea further, we can find the best separation value for the second, third, etc., iteration, until the best node is found.

Table 4. Statistical sub-tree separation over 1000 trees

Separation value	Tree count
23	0
24	1
25	6
26	30
27	88
28	208
29	374
30	509
31	475
32	325
33	167
34	61
35	21
36	7
	2272



Note: the total count of the game trees is higher than 1000 because many values are overlapping – the same X value could divide different trees and the same tree could be divided by different X values.

If we take the tree with branching factor 3, we can apply similar techniques for finding the best separation value. In this case, we have triplets [x, y, z] defining the Minimax value of each sub-tree, so we can build the corresponding 3D matrix displaying the total number of game trees with the given triplet.

While searching this matrix, we look for a separation value of X, so that one sub-tree would be greater or equal with X and two other sub-trees would have smaller estimations and therefore, we maximise the number of trees that would be separated after the first method call, so the best move is found after the first iteration.

5.4 Game Tree Analytical Evaluation

In the previous chapter (BNS enhancement through self-training), statistical analysis, which can improve the performance of the BNS algorithm by calculating and applying "good" separation values, was discussed. Therefore, in the development of this idea, we offer another approach, which is based on a fully analytical determination of the best successful separation value generally, for any type of a tree with various structures (Alpha-Beta range, tree width, depth, etc.).

As stated before, we use abstract domain search in our experiments – meaning tree generation with a fixed structure (width / depth) and randomly assigning leaf values based on the uniform distribution within the given range.

In Figure 28, leaf nodes are noted as probabilistic function F_X. Here, our task is to calculate the resulting function starting from the lowest level (leaf nodes) up to the top level (root node).

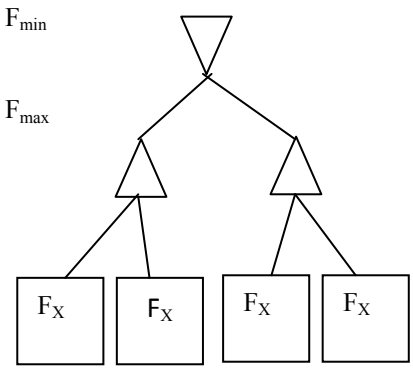


Figure 28. Application of probabilistic function to maximum and minimum levels

In this case, the following functions demonstrate the behaviour of leaf nodes:

- *Probability density function* describes the relative likelihood for this random variable to occur at a given point. For our example (leaf node values are in interval [0; 80]), this likelihood is given in Figure 29;
- *Cumulative distribution function* describes the probability that a real-valued random variable X with the given probability distribution will be found at a value less than or equal to X. For our example, it is given in Figure 30.

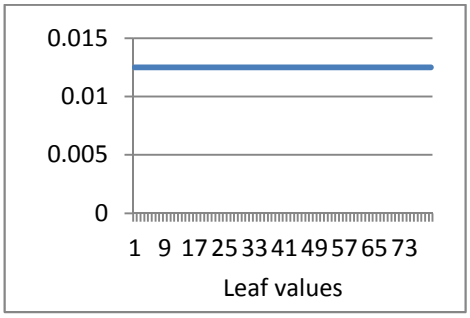


Figure 29. Probability density

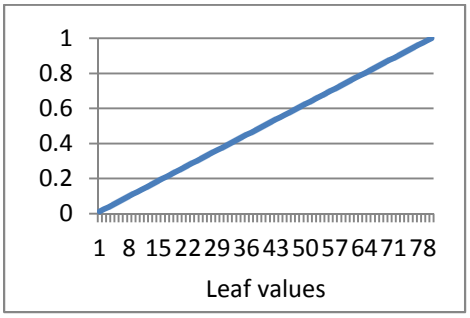


Figure 30. Cumulative distribution

To calculate the probabilistic values correspondingly at maximum and minimum levels, we propose the following formulas based on probability theory, which are applicable for a binary tree (square of probabilistic function). For the max level, it is the probability that both sub-trees are less than our cumulative distribution function and for the min level, it is the probability that both elements are not greater than our cumulative distribution function:

$$F_{max} = (F_x)^2 \qquad (12)$$

$$F_{min} = 1 - (1 - F_x)^2 \qquad (13)$$

For trees with a larger branching factor, the following general formula should be used, where *w* is the width of the tree:

$$F_{max} = (F_x)^w \qquad (14)$$

$$F_{min} = 1 - (1 - F_x)^w \qquad (15)$$

Correspondingly, if we apply this formula to our example with a binary tree with leaf nodes in the given range [0;80], we receive the following equations:

$$F_{max} = \left(\frac{x}{80}\right)^2 \qquad (16)$$

$$F_{min} = 1 - \left(1 - \frac{x}{80}\right)^2 \qquad (17)$$

By using these formulas we can build up the following matrix (Table 5) with iteration results and iteration values for each minimum and maximum level up to the

level of depth 14 (actually, we start from the lowest level with leaf nodes and go up to the highest level – the root node).

Table 5. Calculated cumulative distribution for binary tree with leaf node values from interval [0; 80] and depth 14

Leaf values		*Level*				
		1 – min	2 – max	3 – min	…	14 – max
x	F_x	$1-(1-F_x)^2$	$(F_x)^2$	$1-(1-F_x)^2$	…	$(F_x)^2$
1	1 / 80	0,02484375	0,00061721	0,00123404	…	0
2	2 / 80	0,049375	0,00243789	0,00486984	…	0
3	3 / 80	0,07359375	0,00541604	0,01080275	…	0
…	…	…	…	…	…	…
80	80 / 80	1	1	1	…	1

Figure 31 demonstrates the progress of the cumulative probability function bottom up, changing its slope and becoming nearer to vertical. Correspondingly, the transformed probability density function is displayed in Figure 32 with higher and higher peaks at each subsequent level, where the highest peak corresponds to level 14.

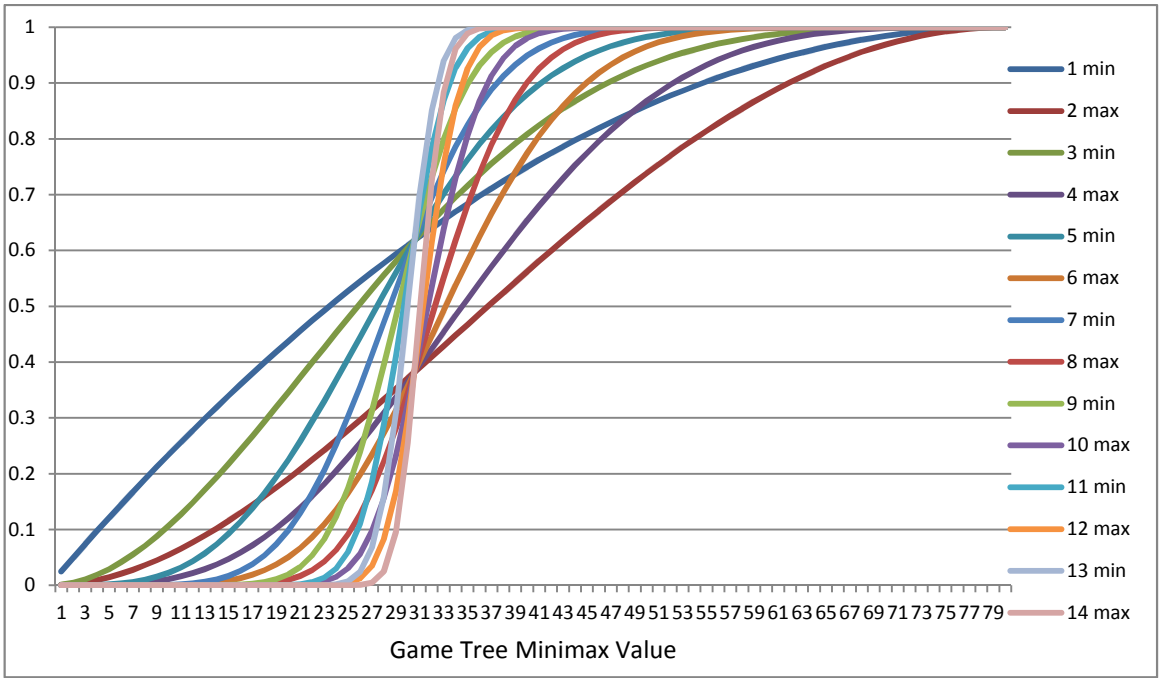


Figure 31. Cumulative probability function by level for depth 14

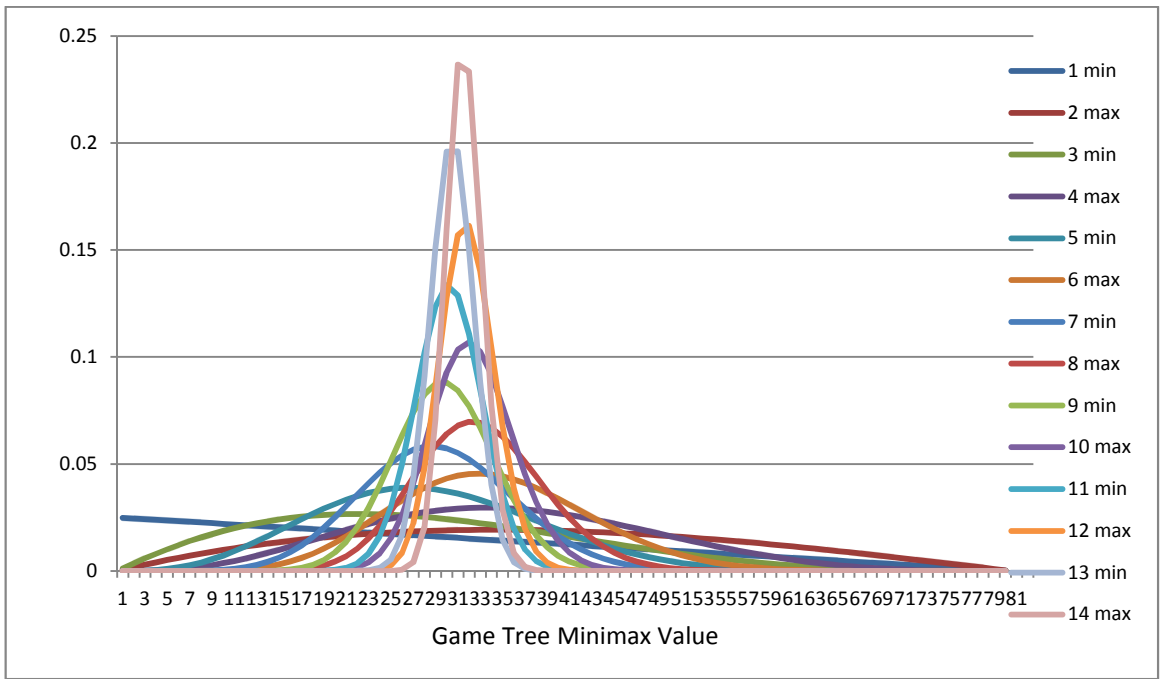


Figure 32. Probability density function by level for depth 14

In the conducted experiments, the statistical information is collected to prove the correctness of the analytical game tree evaluation. The difference between the analytically derived data and statistical experiments is shown in Figure 33. The error rate is relatively low, meaning that the analytical estimation is really close to the experimentally derived results.

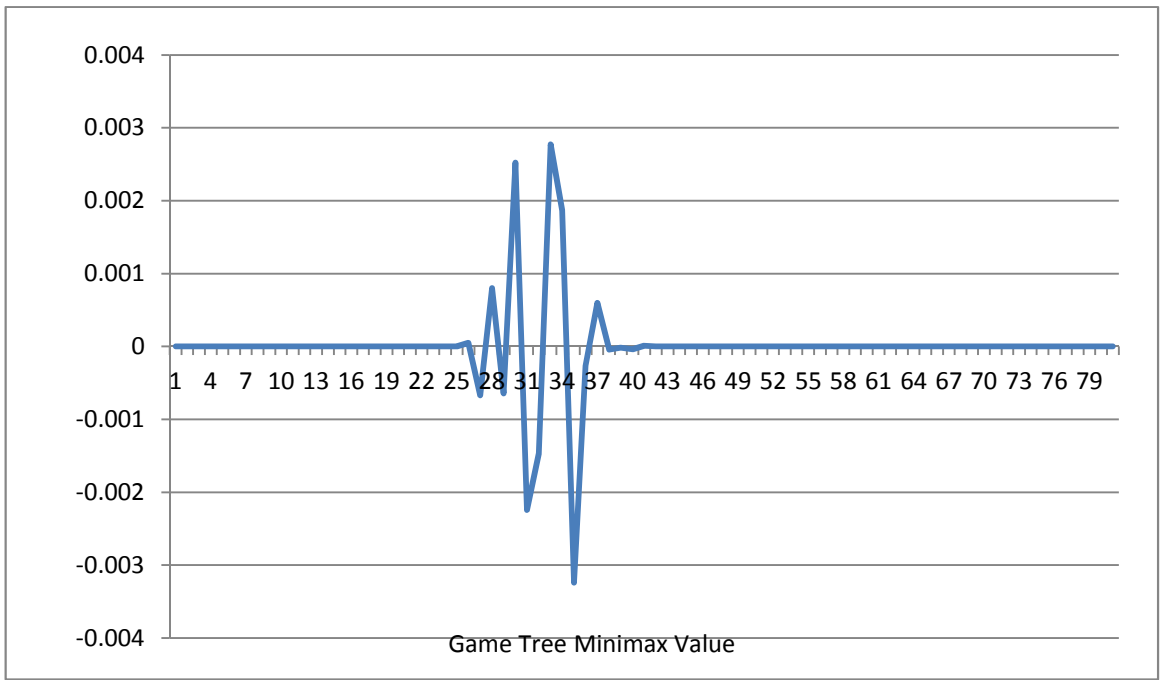


Figure 33. Error function between analytical estimation and experimental results

The resulting probability density function is given in Figure 34. These results correspond to the statistically derived results in the previous section (Table 2).

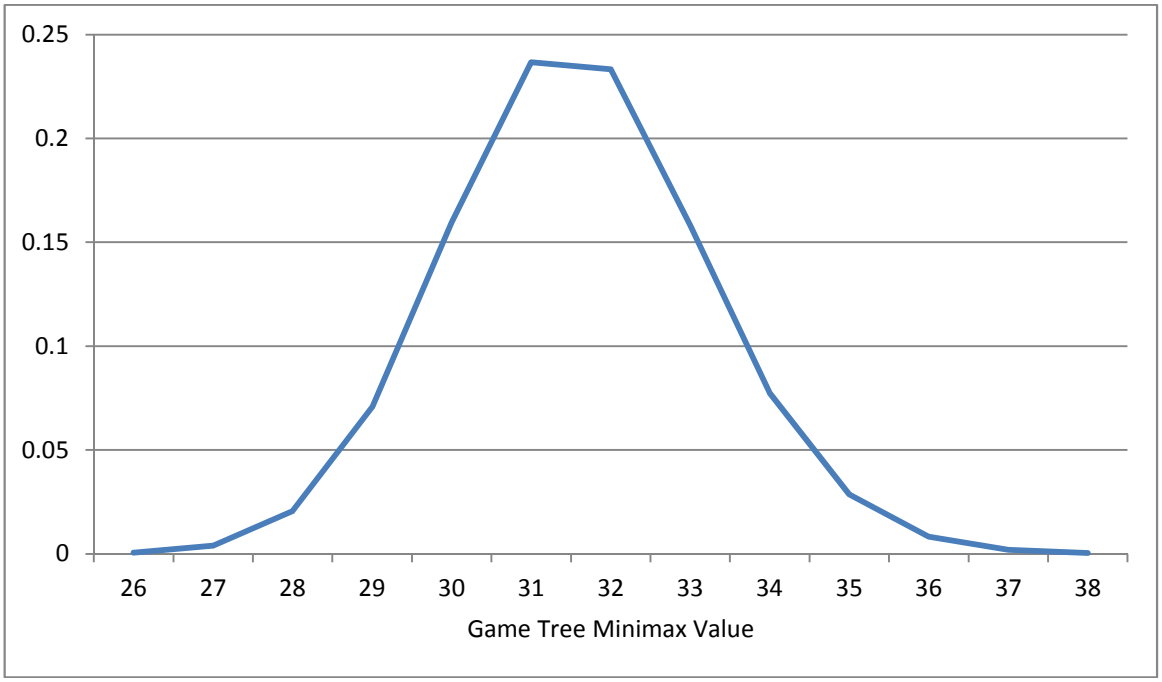


Figure 34. Resulting zoomed-in function

Given the probability density function, we can predict the most probabilistic outcome of the game tree. Thus, we can choose the best separation value for our BNS algorithm – a value of X such that the greatest number of trees will be separated / divided after the first iteration of the algorithm.

These are the same values as we used in the statistical evaluation before, except that analytically we could improve the precision and make calculations much faster without performing long-running experiments. However, we should note that this is valid for trees with uniform distribution only. Position evaluations in many real life games do not follow uniform distribution, and it could be more complex to construct accurate analytical evaluation for them.

We are querying our tree with some separation value X. Therefore, for a given density function, we can calculate the probability that the tree value is less than our test value, or that the tree value is greater. Thus, our task is to maximise our chances to separate the tree with the given value of X.

For optimization process of finding accurate separation values we use elements of information theory, which relies on amount of information that is missing before

reception. The entropy H, of a discrete random variable X, is a measure of the amount of uncertainty associated with that value of X [53].

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i) \quad (18)$$

Having a probabilistic outcome when the tree is separated with probability P and its counterpart outcome when the tree is not separated with probability 1-P, results in binary entropy function H_b [53]. The entropy is maximised at 1 bit per trial when the two possible outcomes are equally probable, as in an unbiased coin toss.

$$H_b(p) = -p \log_2 p - (1-p) \log_2 (1-p) \quad (19)$$

Therefore, we should find such a separation value that maximises the amount of information received after querying the tree. For the first iteration, we receive value 30. For the second iteration, we do the same procedure, i.e., if separation is not obtained after the first query, that means all sub-trees are either less (fall down) or greater (fall up). Thus, we chose the next separation value in the given range maximising the probability of successful separation. Correspondingly, the separation values for the second iteration are 29 and 31, respectively.

In Figure 35, separation value X_1 is shown for the first iteration. If no successful separation is obtained after the first query, then we use the next group of separation values X_2, going to the left or to the right.

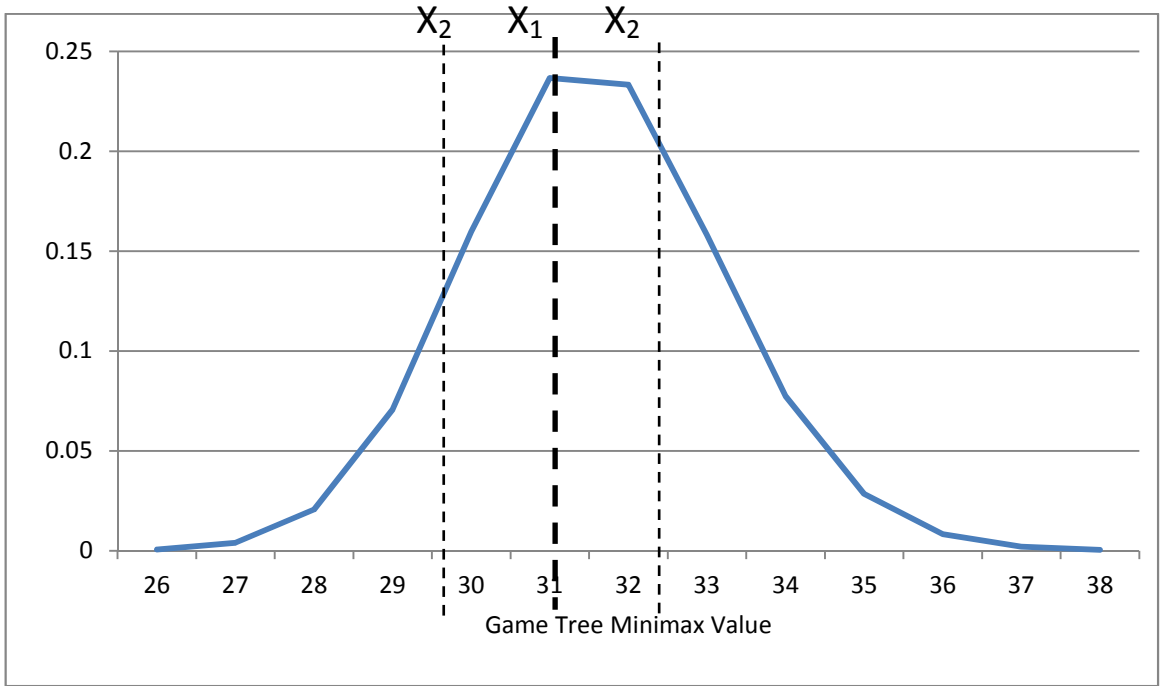


Figure 35. Separation value obtaining with help of density function

Similarly, we seek separation values for a third and a fourth, etc., iteration until the best value is found. At each step, we reduce the Alpha-Beta window. This process is similar to a binary search, except for the selection separation coefficients, where we use the probability density function.

5.5 Experimental Results

The following algorithms were implemented during this research – Alpha-Beta, NegaScout, NegaC*, SSS*, Dual*, MTD(f) and BNS. Versions both with and without transposition tables (TT) were used in our setup.

These algorithms were tested in an abstract domain – generating the game tree test set with a fixed structure (width / depth) and randomly assigning leaf node values from the given range. Then, these experiments were extended to trees with a different branching factor, starting from 2 to 5 and a full alpha-beta window (unlimited range [-infinity, +infinity]).

All the algorithms were run on the same game tree test set (each consisting of 10 000 generated samples) to compare the algorithm efficiency under the same conditions. For each algorithm, the number of visited leaf nodes (evaluation function call) and the total number of visited nodes was measured and the average number per

tree was calculated. In most cases, the first parameter is more important, because in real games evaluation, functions are usually complex enough and require some computing resources. The second parameter is usually less important but for some algorithms, the total number of nodes increases dramatically and should be considered when comparing algorithm efficiency. In the algorithms with reiterative techniques based on transposition tables, when the node is visited multiple times, the total number of nodes is increased and the number of leaf nodes remains the same, as this information is stored in transposition table.

In the chart in Figure 36, the MTD(f) performance is taken as the base point (treated as 100%) and the performance of other algorithms is measured as a ratio to this, i.e., a result greater than 100% means a larger number of search iterations and respectively, only BNS was able to show results less than 100%. It is a combined graph showing trends increasing the width of the search tree – from binary tree to a tree with 5-width structure at each node. In this chart, the number of visited leaf nodes is counted.

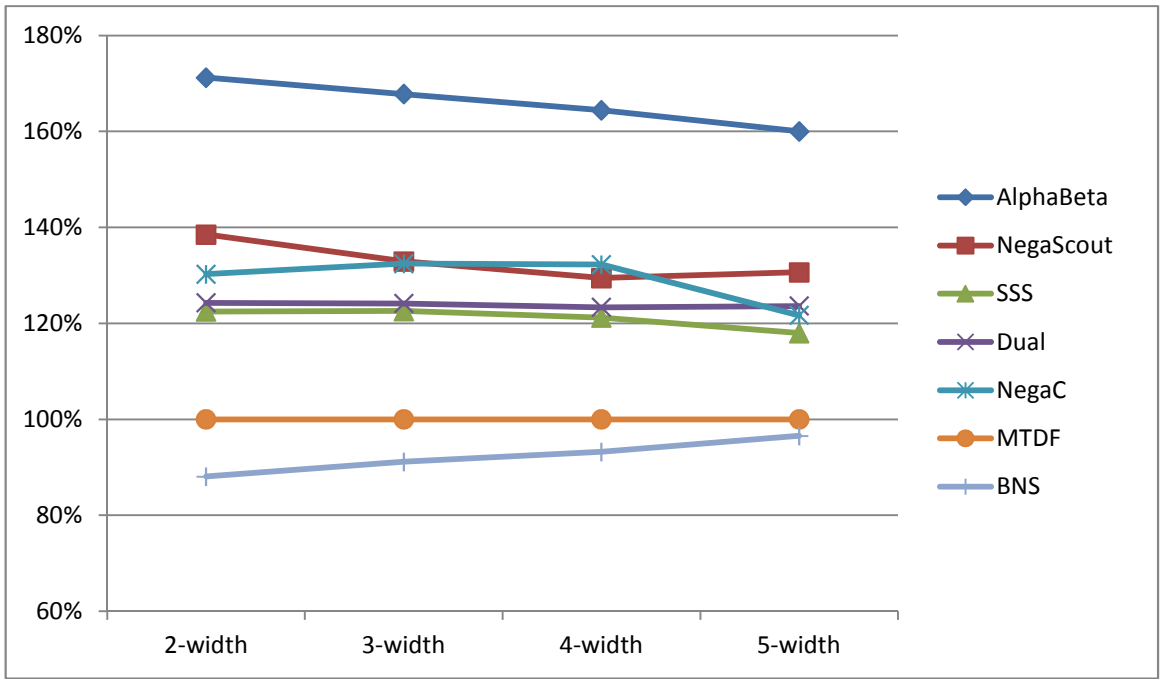


Figure 36. Algorithm's relative performance across different tree widths (leaf nodes visited)

The BNS algorithm shows progress from 88% at the 2-width stage to 96% at the 5-width stage.

Figure 37 demonstrates the same data slice but here, the total number of visited nodes is measured. It can be seen that the BNS performance still remains at approximately 80% compared with the MTD(f) algorithm across all branching factors. Note: SSS and Dual algorithms show low results of 700% and 300%, correspondingly and fall outside of the diagram's range.

Detailed information on the performance of the algorithms can be found in Appendix A.

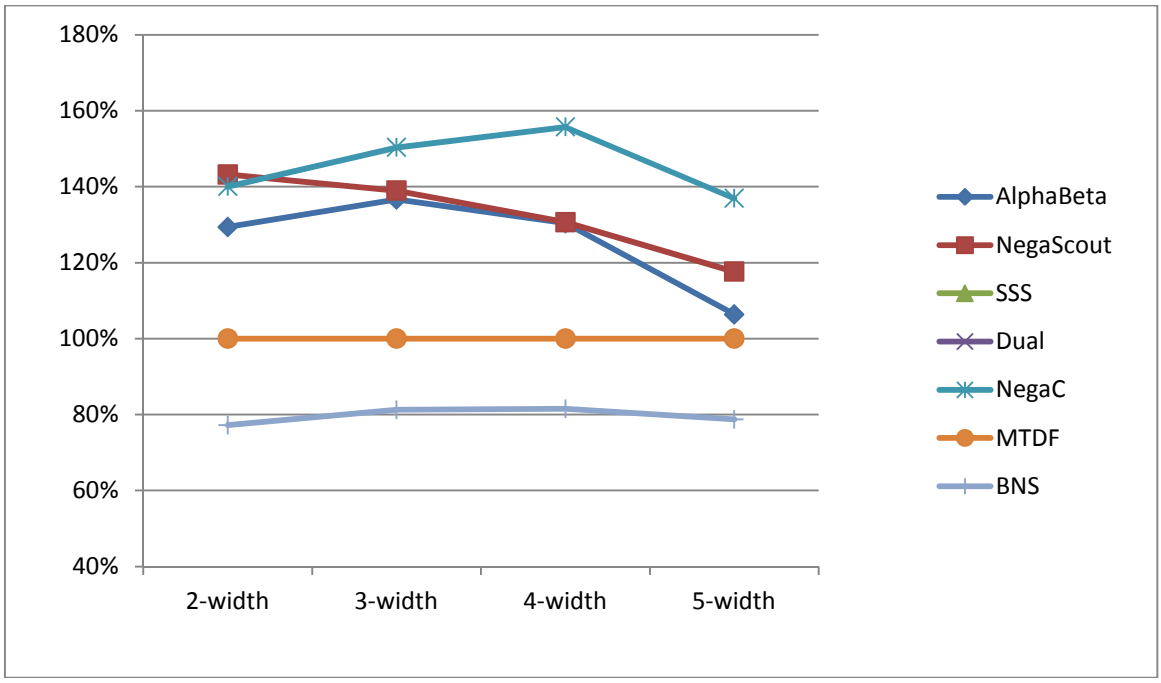


Figure 37. Algorithm's relative performance across different tree widths (total nodes visited)

5.6 Conclusions and Future Work

The main goal of these experiments with abstract domain games was to show that it is possible to find the best move without the exact tree Minimax value. After self-training based on multi-dimension statistics, the proposed BNS algorithm was able to demonstrate better results than other existing algorithms. Game tree analytical evaluation gives additional improvement allowing us to use this algorithm as a general-purpose approach for different tree types.

Having analysed the results we can conclude the following:

- Among algorithms without Transposition Tables, BNS shows competitive results. Both the number of leaf nodes and total number of nodes visited is fewer compared with other algorithms;
- The algorithms based on the Transposition Table approach show different performance in different conditions. So far, MTD(f) was the preferred choice providing the highest performance. However, in the current experiments, BNS was demonstrated to be more efficient comparing both the number of scanned leaf nodes and the total number of nodes visited;
- Considering leaf nodes visited, the BNS algorithm demonstrates an improvement in a range from 12% (for binary trees) to 4% (for 5-width trees) compared with MTD(f);
- Regarding the total nodes visited, the BNS algorithm demonstrates a stable improvement of up to 20% across different branching factors compared with MTD(f);

The current results are based on experiments in the abstract domain and additional research is needed to verify the behaviour of the algorithm for wider trees (with branching factors larger than 15–20, which is typical for real games). Interesting results may be obtained in testing non-regular trees with asymmetrical structure.

The next chapter will focus on analysing the algorithm's performance in real domain games.

6 FUZZIFIED TREE SEARCH IN REAL DOMAIN GAMES

In this chapter, we present the experimental results in real domain games, in which the proposed algorithm demonstrated a 10% performance increase over the existing algorithms.

6.1 Game Setup

For our research purposes, we were seeking a non-complex game (like Chess or Go) but on the other hand, the selected game should not be so simple that it could be completely computable.

Thus, for our experiments we have chosen the “Hey! That's My Fish!” game, which is the perfect match for our needs – it is simple but with some subtle strategy behind it [14].

“Hey! That's My Fish!” is a 2–4 player board game. The aim is to collect as many fish as possible with your penguins.

Setup: 60 hexes are randomly laid out in 8 rows, alternating between 7 and 8 hexes each - Figure 38. They are all face up so that you can clearly see where the fish clusters are. Each hex has either 1, 2, or 3 fish on it. There are 30 "1" fish hexes, 20 "2" fish hexes and 10 "3" fish hexes, which results in 100 points (fish) as the maximum possible to get in one game. Each player then places 2–4 penguins (depending on the number of players). They place these one at a time and they must be placed on "1" fish hexes.

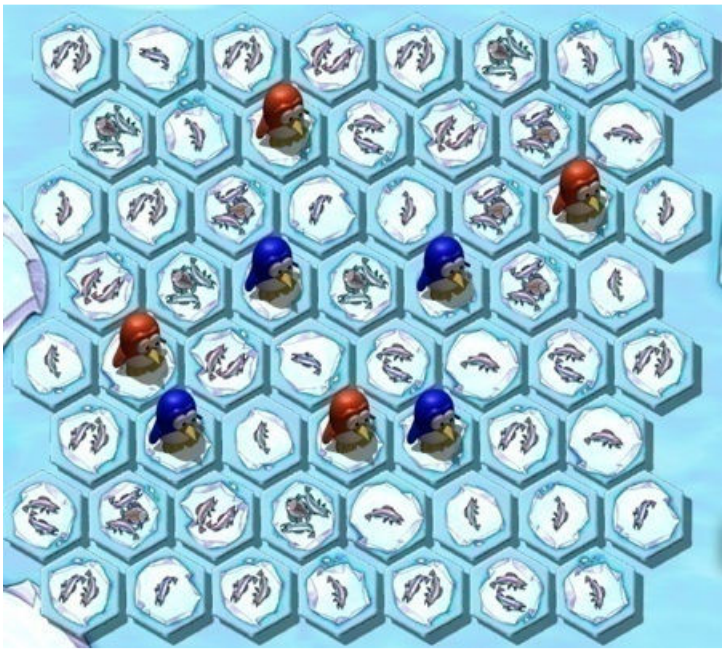


Figure 38. “Hey! That's My Fish!” game board [14]

- Two Player game: each player has 4 penguins
- Three Player game: each player has 3 penguins
- Four Player game: each player has 2 penguins

Play: on his turn, a player moves one of his penguins. He must move it in a straight line and may move it as little as 1 hex and as many hexes as is legal. Penguins must stop before they reach: the edge of the board; a break in the hexes; or another penguin. After moving his penguin, the player then picks up the hex from which his penguin started.

Ending the Game: a player leaves the game when he cannot move any of his penguins. (He takes the last hexes that his penguins are standing on.) When all the players have finished moving the game is over. (Practically, the game actually ends when each player can see that his penguins are each on their own "islands" of ice. Each player then picks up any hexes on these islands, which he could reach.) The players then count all their fish and the player with the most fish wins.

However, there is also a fair amount of tactical content in the game. The moves are not entirely obvious. You can be very aggressive in the game, making clever blocking moves and carefully analysing the vector-based movement [15].

The main strategic element of gameplay is working to isolate other players' penguins, trapping them in small areas; when this happens and an area is isolated that

contains only one penguin, the owner scores all tiles in the isolated area and the penguin is removed from the play. Maximizing the score by getting high-fish hexes is a secondary but important strategic consideration [16].

6.2 Experimental Results

The entire framework was created for the given game and the following algorithms Alpha-Beta, NegaScout, NegaC*, MTD(f) and BNS, were implemented and tested in the real domain. Now we are presenting some additional details on the implementation.

6.2.1 Evaluation Function

To perform the game tree search, a straightforward evaluation function was implemented. At each step we count the amount of fish consumed so far, so an overall evaluation function is the amount of fish obtained by the current player, minus the amount of fish obtained by the opponent, limited by search depth.

$$Evaluation\ function\ = \\ Fish\ Amount\ (player) - Fish\ Amount\ (opponent) \quad (20)$$

The main advantages of this approach are the following: it is fast, simple, reasonable and it converges to correct estimation towards the end of the game. The downsides of this function are that it does not address specific aspects of the game, such as strategic area isolation and opponent blocking but we are mainly focused on search algorithm comparison, which gives us a wide area for research.

Nevertheless, the current implementation achieves a reasonable level of play provided by the program. It allows a 6 ply search depth in the beginning of the game, 10 ply in mid-game and reaching a search depth of 14 ply in the end-game. For details, refer to Figure 39. This search depth generally depends on the decreasing number of moves available during the game. There are approximately 30–50 moves in the initial stage, around 10–30 in the mid-game and less than 10 moves available by the end-game. The game typically terminates in 40–50 total moves.

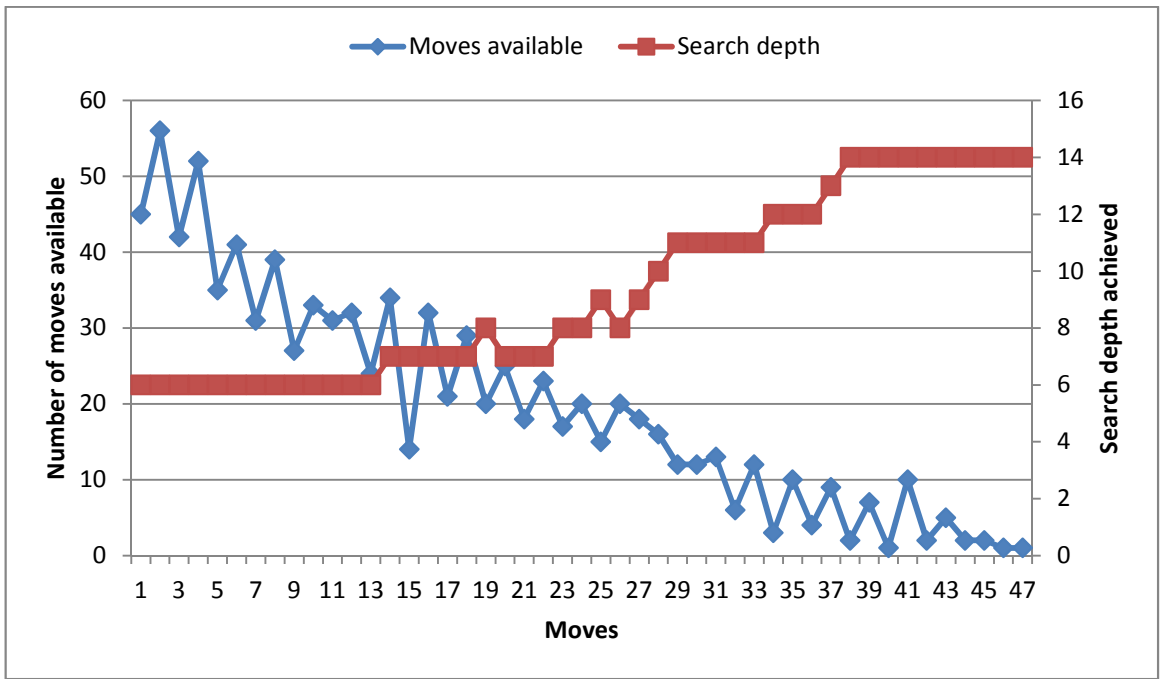


Figure 39. The number of moves available vs. achieved search depth

6.2.2 **Iterative Deepening**

Both the MTD(f) and BNS algorithms can benefit from good estimation of possible outcome. MTD(f) uses a first guess and BNS uses a separation value as their initial seeds. Both goals can be reached efficiently with the iterative deepening technique, i.e., the strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration.

At each iteration, the MTD(f) algorithm returns the exact game tree value, so next time it uses the exact values from the previous step (from previous search level). On the other hand, the BNS algorithm returns a lower bound estimation of the game tree, which means that the exact value could be higher. Nevertheless, as can be seen in the figure, this prediction is successfully used in consequent searches. It should be taken into consideration that the evaluation function is symmetric in its nature, so for an odd search depth, the estimation from the previous odd depth should be used and for an even search depth, the estimation from the previous even depth should be used.

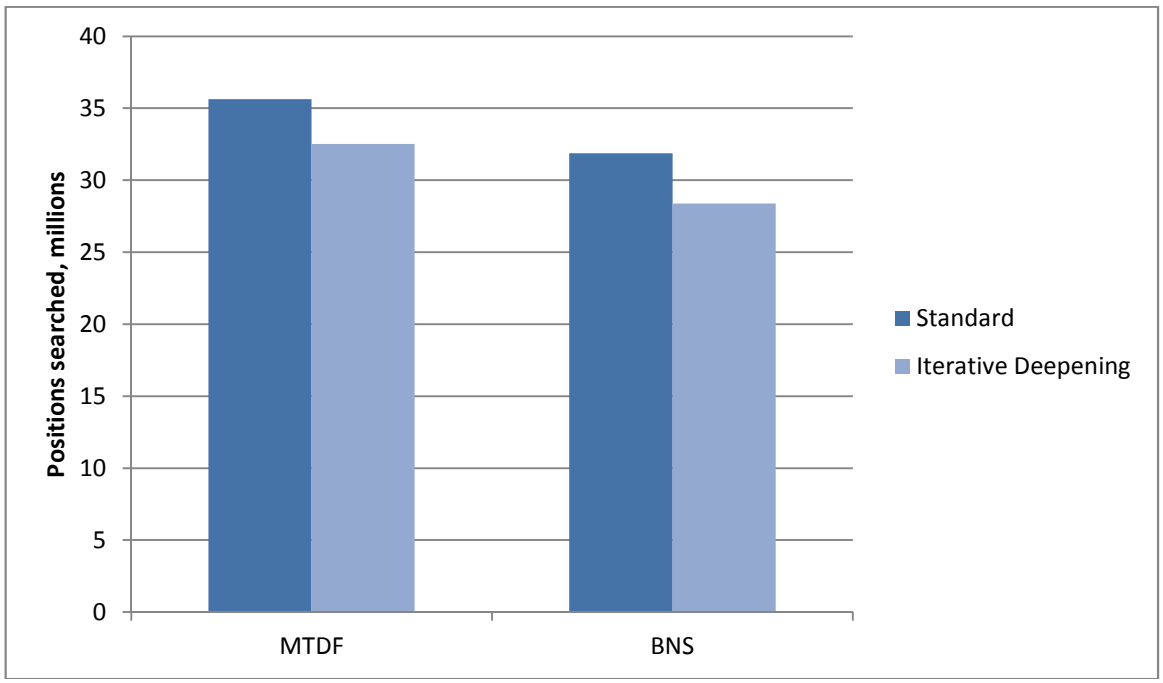


Figure 40. Performance improvement with Iterative Deepening

Figure 40 shows the total number of positions searched (leaf nodes visited / evaluation function calls) by both algorithms during the entire game. It can be seen that introducing the *iterative deepening* technique brings an overall improvement of 9% to 11% for each algorithm and these implementations were used in the further experiments.

6.2.3 **Transposition Tables**

Many algorithms are based on re-iterative search approaches and for example, MTD(f), BNS and others, can speed up their search by using transposition tables. However, in our experiments versions without transposition tables were used, as generally, the implementation of transposition tables requires high memory consumption and our main focus was to verify algorithm behaviour in real domain games.

As we have seen in experiments in the abstract domain, several algorithms, including SSS* and Dual* are usually efficient with transposition tables only and therefore, they were not used in our experiments.

6.2.4 **Algorithm Performance Comparison**

These algorithms were implemented and tested with the “Hey! That's My Fish!” game. The results of one particular match are shown in Figure 41. Despite the fact that the initial board position is random each time, different match series produce quite similar results.

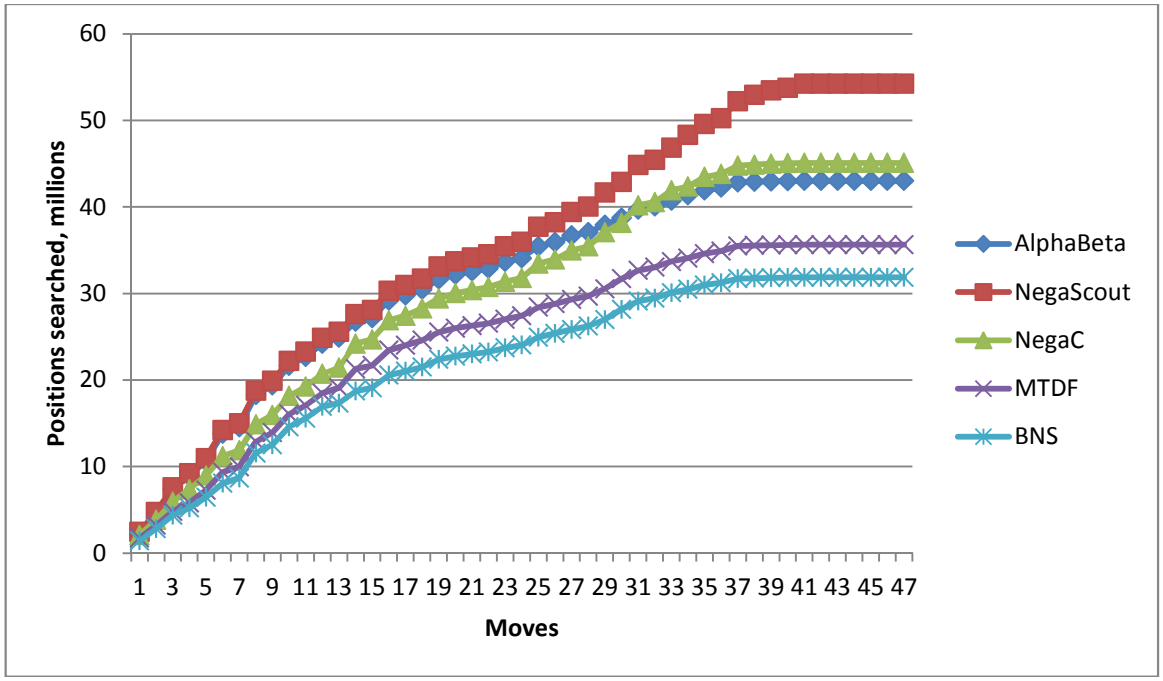


Figure 41. Number of positions searched within the game

The total number of positions searched (number of evaluation function calls / leaf nodes visited) was collected and analysed during the game. This figure presents cumulative statistics, meaning that the total number of leaf nodes is counted during the game, which increases from the initial stage of the game to the end. This particular game has 47 moves in total.

Figure 42 demonstrates the same data but from another perspective. In this chart, the MTD(f) performance is taken as the basic point (treated as 100%) and all others algorithms are measured as a ratio to it.

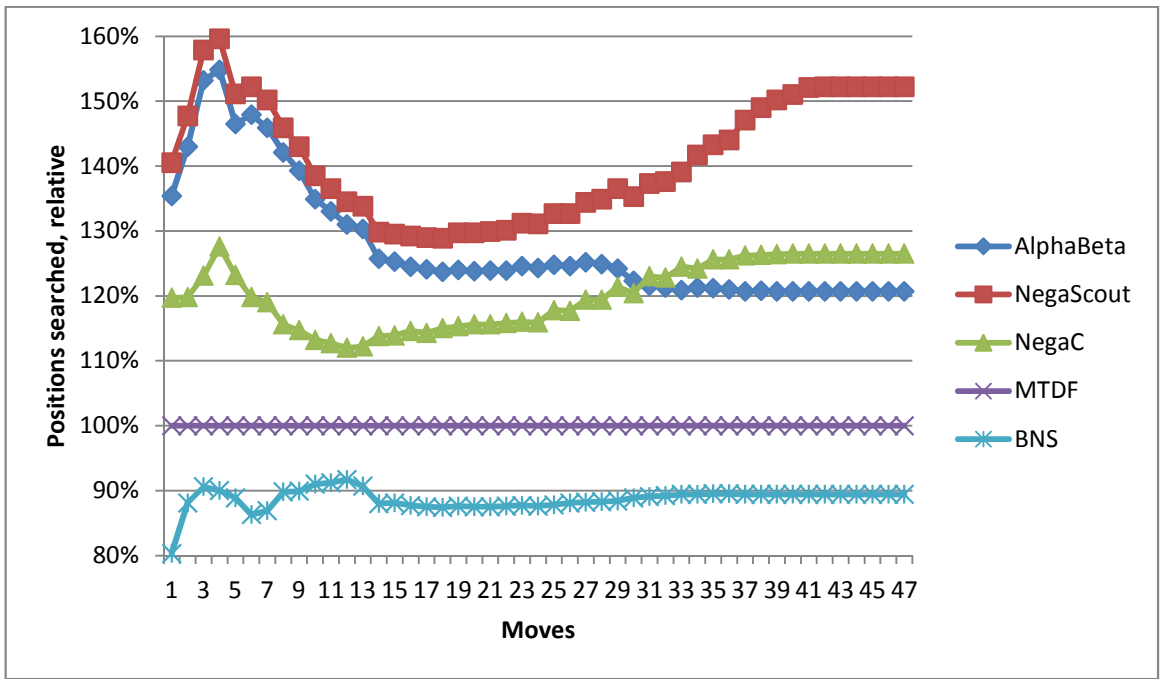


Figure 42. Relative number of positions searched within the game

In these diagrams it can be seen that the NegaScout and Alpha-Beta algorithms have a similar performance during the first half of the game but towards the end, the efficiency of NegaScout is decreased. Similarly, NegaC* shows better results during the initial phases of the game but again, performance also decreased at the end. Apparently, the main reason is that NegaScout / NegaC* are more effective with high branching factors and less effective when the average number of moves is decreasing. Among these algorithms MTD(f) has best results.

On the other hand, the proposed BNS algorithm demonstrates a performance improvement of approximately 15% in the first three moves, then goes a bit below 10% and converges to a stable 10% speed up at the end of the game.

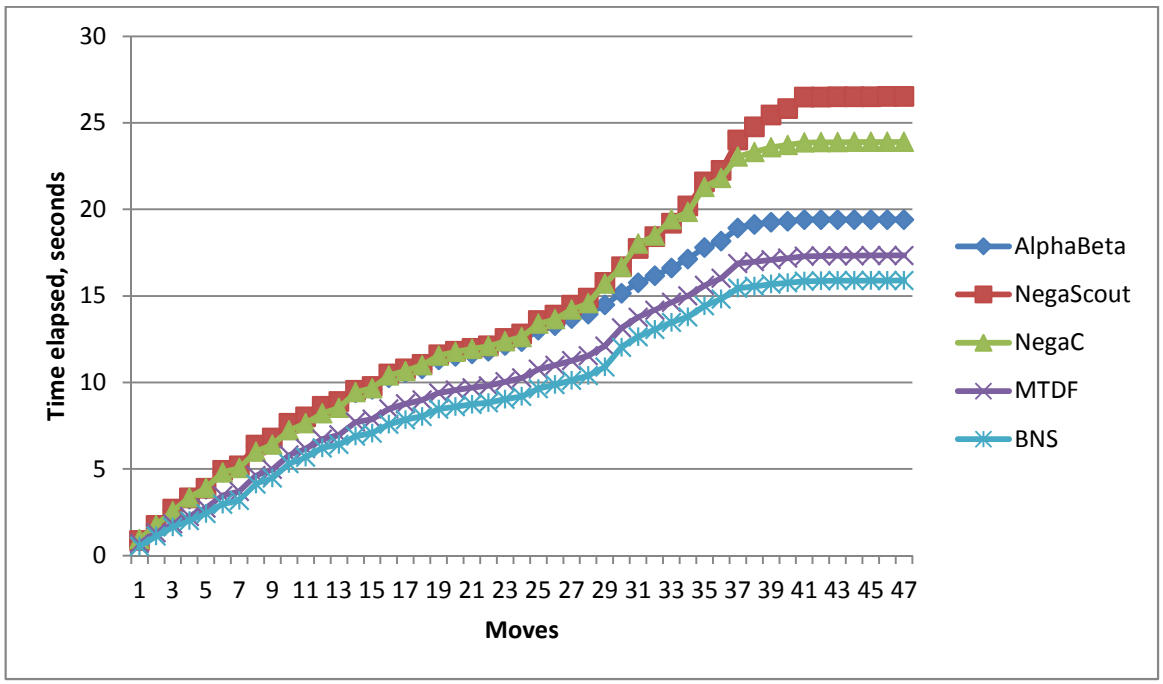


Figure 43. Total time elapsed within the game

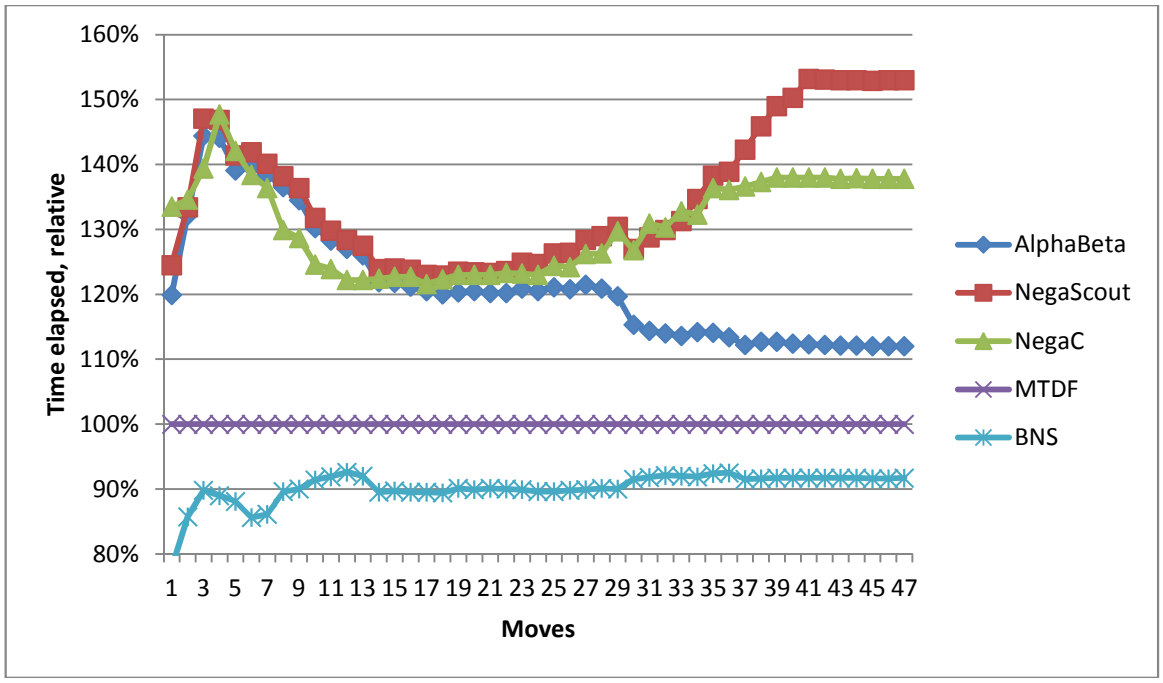


Figure 44. Relative time elapsed within the game

Figure 43 and Figure 44 demonstrate the total time (in seconds) and relative time (measured as a ratio to MTD(f)) elapsed by the algorithms. As can be seen, the BNS algorithm introduces practically no computational overhead compared with others.

6.3 Conclusions and Future Work

The main goal of this research was to show that the proposed BNS algorithm could also be efficient in real domain games. The experiments demonstrate that BNS gives a 10% performance improvement over the MTD(f) algorithm, which is comparable with expected results achieved in experiments in the abstract domain. It can be concluded that BNS demonstrates good potential and could be used as a general-purpose game tree search algorithm.

However, additional research might be needed in the following areas:

- The implementation and analysis of transposition tables, which could potentially increase the performance of the algorithms;
- The use of different knowledge-based or heuristic-based evaluation functions;
- The implementation of a multi-player game. The existing game provides flexible extension up to four players.

Future experiments should also consider analysing algorithm performance in other games but we believe that the proposed approach could be successfully applied for any type of game.

7 FUZZIFIED TREE SEARCH - PRECISION VS. SPEED

This chapter presents a new enhancement to the Fuzzified tree search algorithm for finding nearly optimal solutions.

We propose the notion of quality of search and thus, we can adjust both target quality and performance accordingly to suit our needs. Our experiments show that by applying this technique it is possible to improve algorithm performance significantly, while keeping the error rate very low, which in turn, does not affect overall playing strength of the program.

We describe the experimental setup and empirical results on search quality and performance obtained in a real game. The chapter is concluded with future research directions.

7.1 Nearly Optimal Solutions

Most game tree search algorithms consider finding the optimal move. That is, given an evaluation function, they guarantee that the selected move will be the best according to it. However, in practice, most evaluation functions are themselves approximations and cannot be considered “optimal”. Besides, we might be satisfied with a nearly optimal solution if it gives us a considerable performance improvement.

We present the approximation-based implementations of the Fuzzified game tree search algorithm. The paradigm of the algorithm allows us to find efficiently nearly optimal solutions, so we can choose the "target quality" of the search with arbitrary precision – either it is 100% (providing the optimal move), or selecting a move which is superior to 95% of the solution space, or any other specified value.

Our results show that in games, this kind of approximation could be an acceptable trade-off. For example, while keeping the error rate below 2%, the algorithm achieved more than a 30% speed improvement, which potentially gives us the possibility to search deeper over the same period of time and therefore, make our search smarter. Experiments also demonstrated a 15% speed improvement without significantly affecting the overall playing strength of the algorithm.

7.2 Quality of the Search

Evaluation functions are approximations and might be imprecise; otherwise, there would be no reason to perform a deeper search. They are not optimal by their nature but it is important how close they are to real utility of the board position and how that is related to the chance of winning.

In many game specific situations, very often the program is required to respond in a limited time. Thereby, it becomes more important to find solutions quickly and the optimality of the solution moves to a secondary role. Therefore, we are interested in techniques that would allow us to control the quality of our search while focusing on performance and overall search speed.

The Fuzzified game tree search algorithm fits this idea very well. That is, given an evaluation function, we can extend our search by introducing an additional parameter - quality of search. In other words, this is a guaranteed probability of finding the move, which is in the top N% of possible moves. For example, the traditional implementation uses maximum quality of the search, finding the best move with 100% probability. However, if we were satisfied with a move in the top 10%, we could easily perform this search.

This target quality logically means that if we choose some level of confidence, for example 95%, this means that the move that is returned by search algorithm would be better (not worse) than all other 95% of possible moves. On the contrary, if we choose the highest level of confidence (100%), the move found would be the best of all possible moves.

The most important part of this algorithm is that it is designed to allow choice in the target quality of the search (your level of confidence) arbitrarily. It may depend on the level of playing strength, time remaining for a search and could always be updated dynamically, because all information about those moves already checked is stored in memory. Therefore, if you choose additional move tuning, only new unexplored parts of the tree would be researched with no loss of time (double search).

Therefore, we propose an additional improvement to the existing algorithm to adhere to the aforementioned observations.

Let us explore the existing algorithm and note the additional changes required to make a nearly optimal search or arbitrary search of quality available.

The following is the same search algorithm listed in Figure 27 except for one enhancement – the addition of the search quality parameter.

```
function BNS(node, α, β, quality)
    do
        test := NextGuess(node, α, β)
        betterCount := 0
        foreach child of node
            bestVal := -AlphaBeta(child, -test, -(test - 1))
            if bestVal ≥ test
                betterCount := betterCount + 1
                bestNode := child
                if expectedQuality ≥ quality
                    return bestNode
        update alpha-beta range
    while not((β - α < 2) or (betterCount = 1))
    return bestNode
```

Figure 45. BNS with quality

The expected quality of search is measured as a probability of randomly choosing a move over all possible moves, which satisfy our search criteria (formula 21).

It depends on the number of nodes checked so far and the total number of nodes. Initially, we use the ratio of "better nodes" amongst checked nodes to calculate the expected number of "better nodes" amongst all sub-trees. Thus, the overall expected quality is measured as the probability of having found the best node.

Therefore, the main idea is to stop our search and return the best move found so far, as soon as we are confident about the quality of our search results. This version of the algorithm is used in the following experiments.

$$ExpectedQuality = \frac{1}{\frac{betterCount}{number\ of\ nodes\ already\ checked} * total\ number\ of\ nodes} \quad (21)$$

7.3 Experimental Results

The proposed BNS algorithm with quality was implemented and tested in the "Hey! That's My Fish!" game, which was described in the previous chapter. Optimisation techniques, such as iterative deepening and others were also used in the following

series of experiments. Now, we present additional details and the results obtained in our study.

We conducted experiments with different values of expected confidence ranging from 100% to 0%. Whereas 100% means that we are looking for the best move without compromise, 0% does not mean that we are acting randomly. Instead, we search for the first move that satisfies the selection criteria and only then return it. In practical experiments, the error rate is measured as the ratio of non-optimal to optimal solutions selected during the entire game and the actual quality is calculated as 1 minus the error rate.

We also measure performance improvement as the total number of nodes searched during the entire game, divided by the number of nodes required to search for the highest level of confidence (100% quality).

As seen in Figure 46, the actual quality of the search remains 100% for the first iterations (for expected quality 100%, 90% and 80%, respectively). That means for this particular case, we receive a 20% performance improvement while the algorithm is still able to find the best moves all the time. Then, as the actual quality decreases the performance increases considerably. For example, while retaining 98% quality we achieve a 30% speed improvement and with 95% quality, we achieve a 40% speed improvement. Our diagram terminates at around 75% performance improvement while retaining 75% search quality.

An additional parameter we are tracking is the average error per wrong move found (in game points) – the difference between the evaluation of the best move and the move returned by our algorithm. Initially, there is no error because the algorithm shows 100% actual quality, then for two cases (70% and 60% expected quality), it stays within 1 point per move error and for remaining experiments, it converges to around 1.2 points of error per move. This is a very important indicator, showing that even when there is an error in the algorithm search, the error is small and that the returned move is very close to the optimal one.

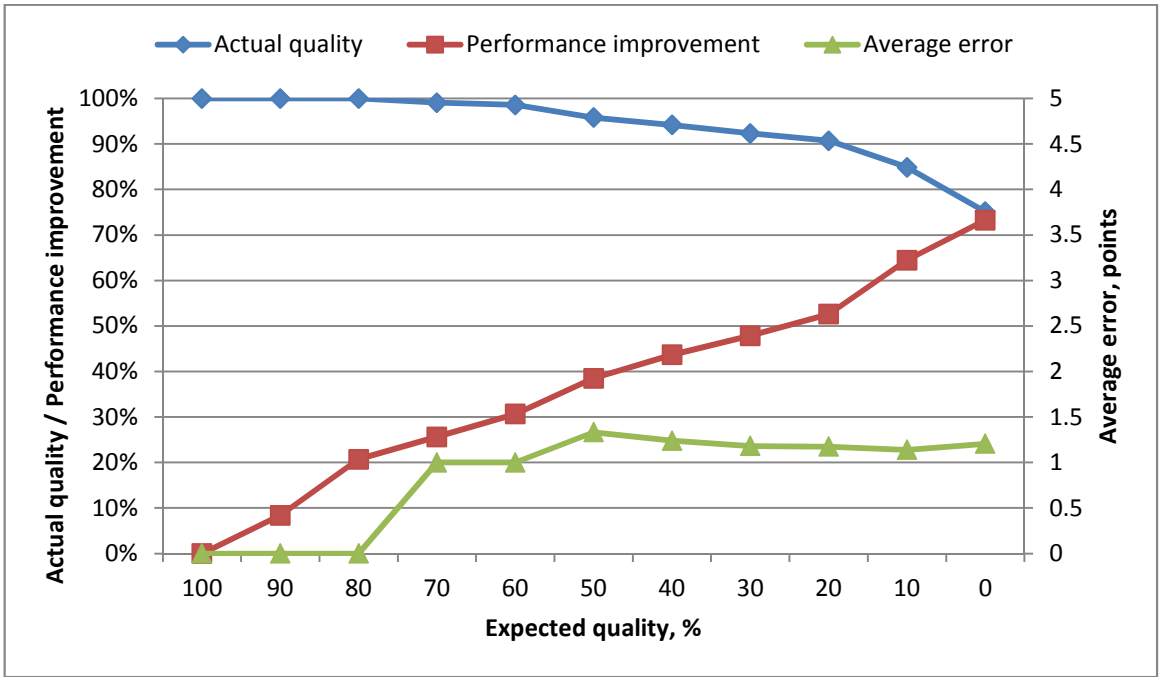


Figure 46. Actual quality vs. Performance improvement. Average error (points)

It is important to note that currently, we operate with a minimal guaranteed (expected) confidence level and thus, actual quality is much higher than the expected values. This is mainly because when searching we are not acting randomly, we are looking for at least one move satisfying our search criteria to guarantee our minimal confidence level.

To conclude, additional experiments have been performed based on a series of games, in order to obtain results that are more objective.

Because "*Hey! That's my Fish*" starts with a random setup and the outcome of the game largely depends on the beneficial location of valuable tiles, we measure algorithm performance based on a series of matches. The program plays 100 matches by itself, calculating the winning probability (number of games won by each algorithm). As player order (who starts first) is also important, we switch players during these series of games, so that the player, who initially started first, starts second. It results in a total series of 200 games for each configuration and the results are displayed in Figure 47.

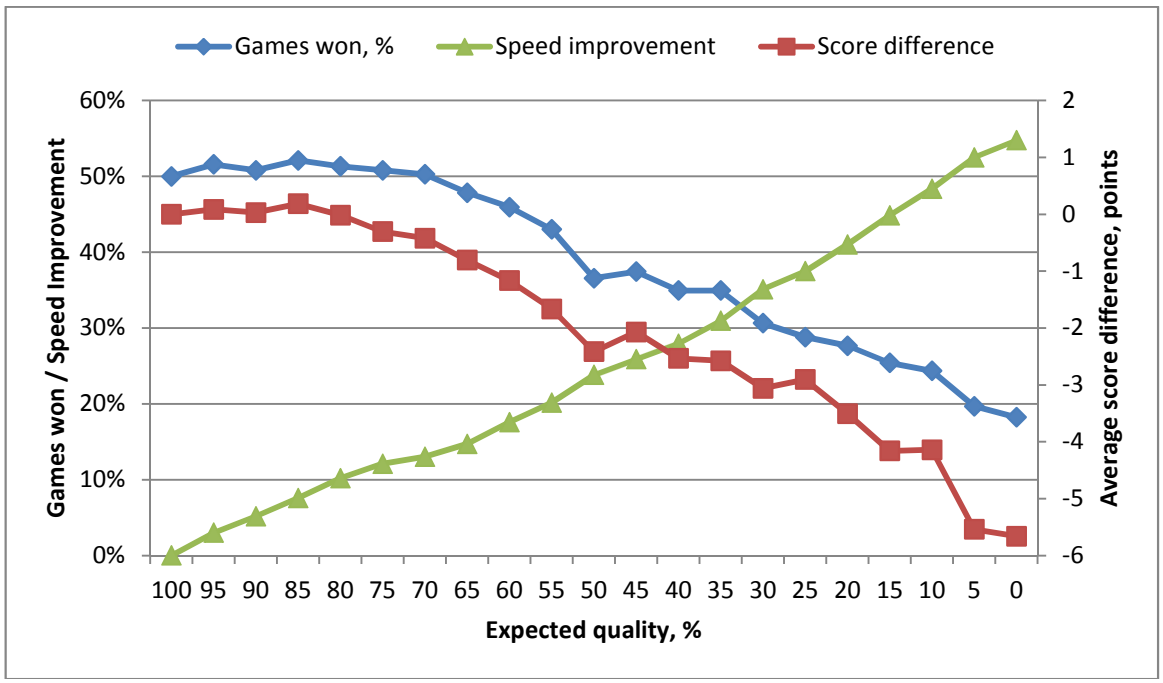


Figure 47. Number of games won (%) vs. Speed Improvement. Average points difference

The following configuration has been used: search depth 6, expected quality of search varying from 100% to 0%. As a reference algorithm (opponent against whom we play), we use the same algorithm but with 100% (maximum) expected search quality.

Performance improvement is measured as the difference between the total numbers of nodes searched by each algorithm during the entire series of games. Score difference represents an average variance in points obtained by each player.

As can be seen, the algorithms start equally having a winning probability of 50% and score difference of 0. As the expected quality decreases, the improvement in speed grows linearly. It is important to note that the winning probability remains close to 50% (with fluctuations of 1–2%) while decreasing the expected quality to 70%. This gives a 15% speed improvement without significantly affecting the overall playing strength of the algorithm. However, decreasing further the expected search quality results in considerable degradation of performance.

7.4 Conclusions and Future Work

The main goal of this research was to show that the Fuzzified tree search algorithm could be easily extended and efficiently used for finding nearly optimal solutions. The

experiments demonstrate a 30% speed improvement over the standard approach while retaining an error rate below 2%. Moreover, in case of error, the selected move is still very close to the optimal solution. Further experiments show a 15% speed improvement without significantly affecting the overall playing strength of the algorithm. It can be concluded that the proposed approximation search paradigm could be used in real domain games with high a level of confidence.

However, additional research might be needed in the following areas:

- improve estimation precision of expected quality vs. actual quality achieved;
- apply different heuristic-based evaluation functions.

Future experiments should also consider analysing the algorithm performance and efficiency in other games but we believe that the proposed approach could be successfully applied to any type of game.

8 CONCLUSION

In this chapter, we will summarise all the items discussed in this thesis.

We have discussed game theory aspects and reviewed the state-of-the-art game tree search algorithms. We also analysed alternative approaches based on probabilistic forward pruning.

We have investigated possible solutions for further improvement of the existing search algorithms; several results were achieved in this area.

Firstly, we have proposed a new approach that allows the best move to be found more quickly while visiting fewer nodes. The main idea is based on the fact that the exact game tree evaluation is not required to find the best move. Therefore, pruning techniques may be applied earlier, which result in a faster search and greater performance. We also have proposed the Fuzzified game tree search algorithm and two improvements based on statistical training and game tree analytical evaluation. Applied to an abstract domain, this algorithm outperformed existing ones, such as Alpha-Beta, PVS, NegaScout, NegaC*, SSS*/ Dual* and MTD(f).

Secondly, we presented experimental results in real domain games, where the proposed algorithm demonstrated a 10% performance increase over existing algorithms. Thus, it can be concluded that the BNS demonstrated good potential and could be used as a general-purpose game tree search algorithm.

Finally, we introduced an enhancement to the Fuzzified game tree search algorithm based on quality of search, which means we can adjust both the target quality and performance accordingly to suit our needs. Important part of the algorithm is that it is designed to allow choosing target level of play arbitrarily, thus easily simulating weaker or smarter player. The experiments demonstrated a 30% speed improvement over the standard approach while retaining an error rate below 2%. Moreover, in the case of error, the selected move was still very close to the optimal solution. Further experiments showed a 15% speed improvement without significantly affecting the overall playing strength of the algorithm. Thus, it could be concluded that the proposed approximation search paradigm could be used in real domain games with a high a level of confidence.

However, there are additional areas of research available.

An important aspect for future research would be to analyse whether Fuzzified game tree search with quality would lead to improvements in playing strength, when given the same amount of time. Thus, by pruning potentially irrelevant nodes in earlier iterations, we get to look deeper by doing additional iterations and thus, hopefully, play better.

Future experiments should consider analysing the algorithm performance in other games but we believe that the proposed approach could be successfully applied for any type of game.

BIBLIOGRAPHY

[1] T. A. Marsland, M. Campbell. Parallel Search of Strongly Ordered Game Trees. ACM Comput. Surv., 1982

[2] Judea Pearl. The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. Communications of the ACM, 1982

[3] A. Reinefeld. An Improvement to the Scout Tree-Search Algorithm. ICCA Journal, 1983, Vol. 6, No. 4, pp. 4-14

[4] A. Reinefeld. Spielbaum-Suchverfahren. Informatik-Fachbericht 200, Springer-Verlag, 1989

[5] Jean Christophe Weill. The NegaC* search. ICCA Journal, March 1992

[6] Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. A New Paradigm for Minimax Search, Technical Report EUR-CS-95-03, 1994

[7] Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. Best-First and Depth-First Minimax Search in Practice, Proceedings of Computing Science in the Netherlands, 1995

[8] Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. An Algorithm Faster than NegaScout and SSS* in Practice, Computer Strategy Game Programming Workshop at the World Computer Chess Championship, 1995

[9] Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de. Best-First Fixed-Depth Minimax Algorithms, Artificial Intelligence, volume 87, 1996

[10] Yngvi Björnsson. Selective Depth-First Game-Tree Search. Ph.D. thesis, University of Alberta, 2002

[11] Russell, Stuart J., Norvig, Peter. Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Pearson Education, Inc., 2010

[12] Dmitrijs Rutko. Fuzzified Algorithm for Game Tree Search. Second Brazilian Workshop of the Game Theory Society, BWGT 2010

[13] Dmitrijs Rutko. Fuzzified Algorithm for Game Tree Search with Statistical and Analytical Evaluation. Scientific Papers, University of Latvia, 2011, Vol. 770, pp. 90-111

[14] Hey That's My Fish - German Board Games
http://www.mrbass.org/boardgames/heythatsmyfish/

[15] Review of Hey! That's My Fish! – RPGnet
http://www.rpg.net/reviews/archive/11/11936.phtml

[16] Hey! That's My Fish! Remarkable Depth for Minimal Rules http://playthisthing.com/hey-thats-my-fish

[17] Kriegspiel (chess) http://en.wikipedia.org/wiki/Kriegspiel_(chess)

[18] Minimax http://chessprogramming.wikispaces.com/Minimax

[19] Negamax http://chessprogramming.wikispaces.com/Negamax

[20] Alpha-Beta http://chessprogramming.wikispaces.com/Alpha-Beta

[21] Principal Variation Search http://chessprogramming.wikispaces.com/Principal+Variation+Search

[22] NegaScout http://chessprogramming.wikispaces.com/NegaScout

[23] NegaC http://chessprogramming.wikispaces.com/NegaC*

[24] SSS* and Dual* http://chessprogramming.wikispaces.com/SSS*+and+Dual*

[25] MTD(f) http://chessprogramming.wikispaces.com/MTD(f)

[26] Larry Harris. The Heuristic Search And The Game Of Chess - A Study Of Quiescence, Sacrifices, And Plan Oriented Play. IJCAI 1975, pp. 334-339

[27] Aske Plaat. MTD(f) - A Minimax Algorithm faster than NegaScout, 1997 http://people.csail.mit.edu/plaat/mtdf.html

[28] Aske Plaat. Research Re: search & Re-search, PhD Thesis, Tinbergen Institute and Department of Computer Science, Erasmus University Rotterdam, Thesis Publishers, Amsterdam, The Netherlands, June 20, 1996

[29] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. Artificial Intelligence, 1975

[30] Victor Allis. Searching for Solutions in Games and Artificial Intelligence. Ph.D. Thesis, University of Limburg, 1994

[31] Computing Dictionary, http://dictionary.reference.com/browse/search+problem

[32] Lewis Stiller. Multilinear Algebra and Chess Endgames. Games of No Chance, MSRI Publications, Vol. 29, 1996

[33] Marc Bourzutschky, Yakov Konoval. "7-man endgames with pawns". CCRL Discussion Board, 2006

[34] Thomas Anantharaman, Murray Campbell, and Feng-hsiung Hsu. Singular extensions: Adding Selectivity to Brute-Force Searching. AAAI Spring Symposium, Computer Game Playing, 1988, pp. 8-13.

[35] Thomas Hauk, Michael Buro, Jonathan Schaeffer. *-Minimax Performance in Backgammon. Computers and Games - CG, 2004, pp. 51-66

[36] H.J. Berliner. The B* Tree Search Algorithm: A Best-First Proof Procedure. Artificial Intelligence 12(1), 23–40 (1979)

[37] M. Buro. ProbCut: An effective selective extension of the alpha–beta algorithm. ICCA Journal 18(2), 71–76 (1995)

[38] M. Buro. Experiments with Multi–ProbCut and a new high–quality evaluation function for Othello. In: Workshop on Game–Tree Search, NECI (1997)

[39] Jiang, A.X., Buro, M. First Experimental Results of ProbCut Applied to Chess. In: Proceedings of the Advances in Computer Games Conference, Graz, vol. 10 (2003)

[40] Björnsson, Y., Anthony Marsland, T. Multi-cut alpha-beta-pruning in game-tree search. Theoretical Computer Science 252(1-2), 177–196 (2001)

[41] Lim, Y.J., Lee, W.S. RankCut - A Domain Independent Forward Pruning Method for Games. In: AAAI (2006)

[42] Chang, H.-J., Tsai, M.-T., Hsu, T.-S.: Game Tree Search with Adaptive Resolution. In: Proceedings of the 13th Advances in Computer Games Conference, ACG 2011, Tilburg, Netherlands (2011)

[43] Dmitrijs Rutko. Fuzzified Tree Search in Real Domain Games. Advances in Artificial Intelligence, Springer, 2011, LNAI, Volume 7094, pp. 149-161

[44] Dmitrijs Rutko. Fuzzified Game Tree Search - Precision vs Speed. PRICAI 2012: Trends in Artificial Intelligence, Springer, 2012, LNAI, Volume 7458, pp. 504–515

[45] Michael Buro. Techniken für die Bewertung von Spielsituationen anhand von Beispielen. Ph.D. Thesis. University of Paderborn, Paderborn, Germany, (1994)

[46] Noam Nisan, Tim Roughgarden, Eva Tardos, Vijay V. Vazirani. Algorithmic Game Theory, Cambridge University Press, 2007, 776 pages

[47] John Maynard Smith. Evolution and the Theory of Games. Cambridge University Press, 1982, 226 pages

[48] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, O. Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. Communications of the ACM, Volume 55, Issue 3, 2012, pp. 106-113

[49] Claude E. Shannon. Programming a Computer for Playing Chess. Philosophical Magazine, Ser.7, Vol. 41, No. 314, 1950, pp. 256-275

[50] Ken Thompson. 6-Piece Endgames. ICCA Journal, Vol. 19, No. 4, 1996, pp. 215-226

[51] Ken Thompson. The Longest: KRNKNN in 262. ICGA Journal, Vol. 23, No. 1, 2000, pp. 35-36

[52] Donald Michie. Game-playing and game-learning automata. Advances in Programming and Non-Numerical Computation, 1966, pp. 183-200

[53] David MacKay. Information Theory, Inference, and Learning Algorithms. Cambridge University Press, 4th printing, 2005, 640 pages

[54] David Eppstein, Strategy and board game programming, Dept. Information & Computer Science, University of California, Irvine, http://www.ics.uci.edu/~eppstein/180a/index.html

[55] Horizon Effect http://chessprogramming.wikispaces.com/Horizon+Effect

[56] Endgame tablebase http://en.wikipedia.org/wiki/Endgame_tablebase

[57] George F. Luger, Artificial Intelligence: Structures and Strategies for Complex Problem Solving, Addison-Wesley, 6th edition, 2008, 784 pages

APPENDIX

Appendix A. Performance Results in Abstract Domain

The following section contains the performance results of the algorithms implemented during the current research in the abstract domain for different tree structures and leaf node ranges.

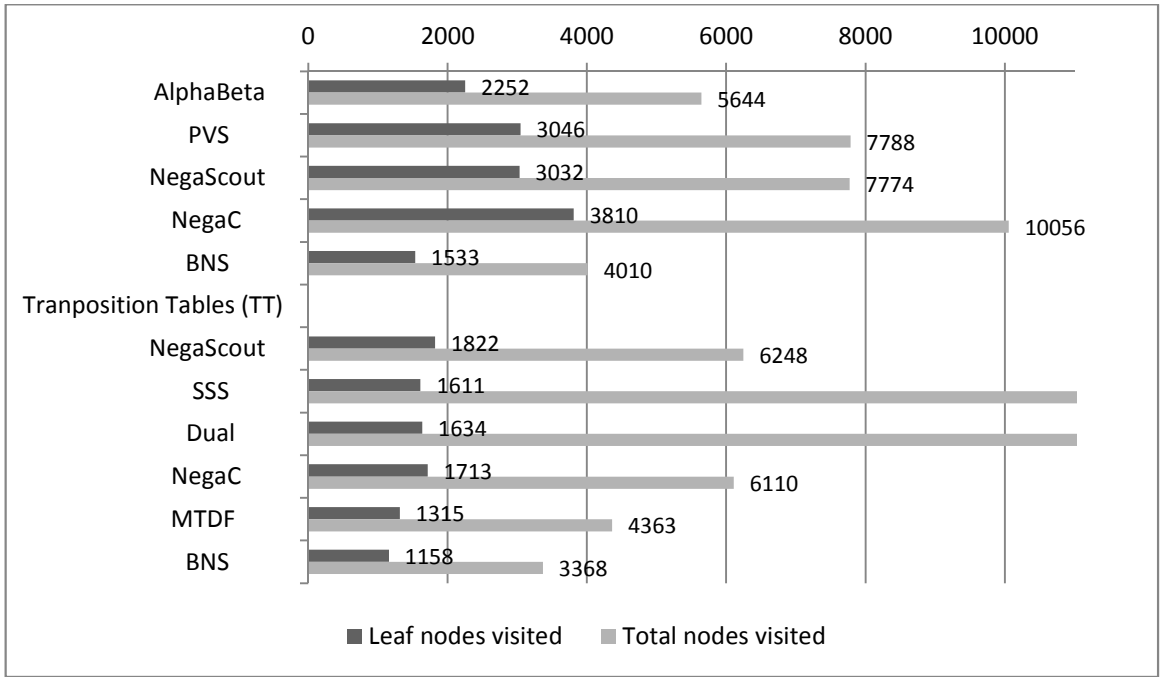


Figure 48. Tree width – 2, depth – 14

Leaf node range 0..80; full alpha-beta window

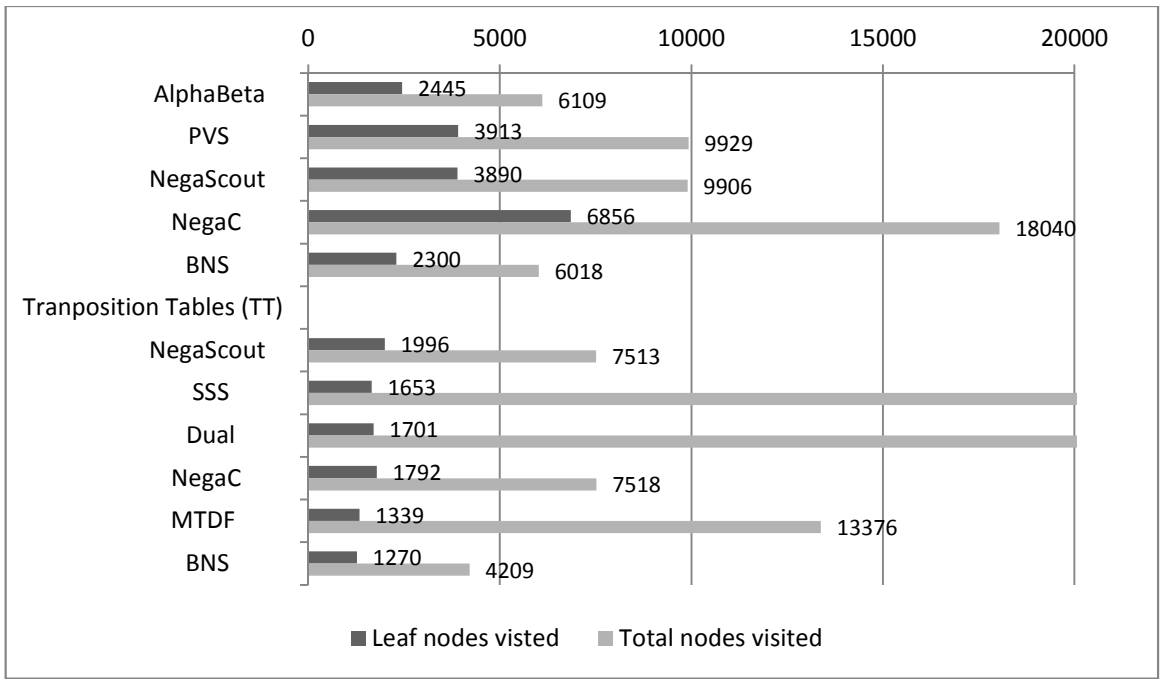


Figure 49. Tree width – 2, depth – 14

Leaf node range 0..800; full alpha-beta window

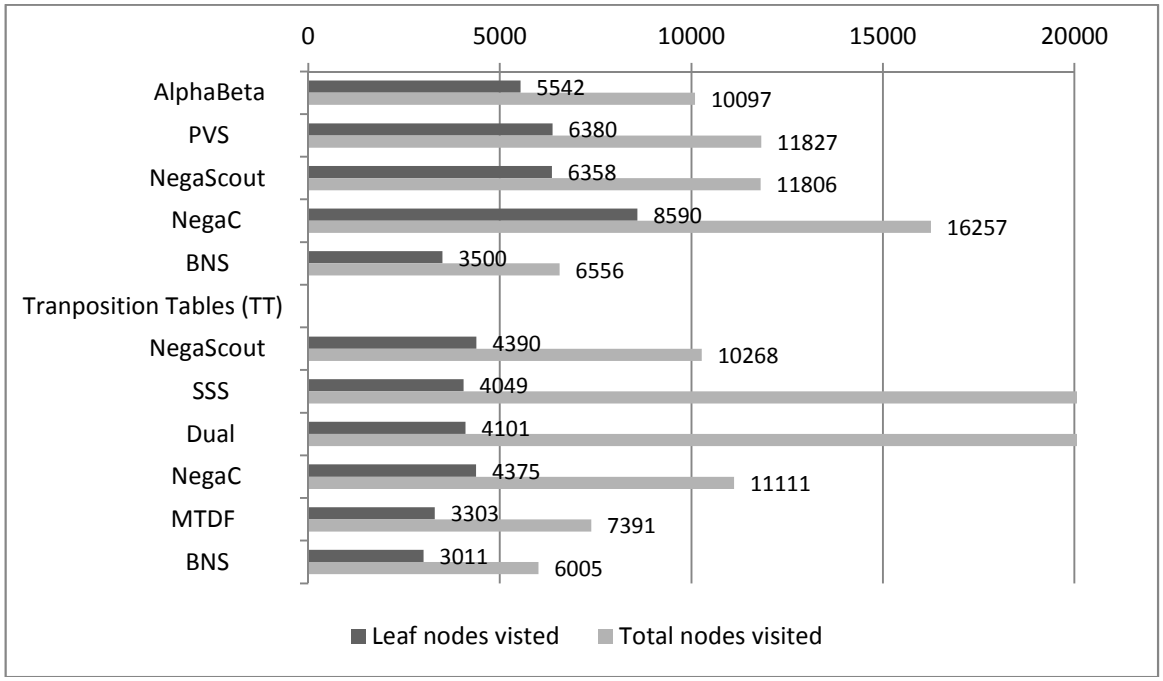


Figure 50. Tree width – 3, depth – 10

Leaf node range 0..80; full alpha-beta window

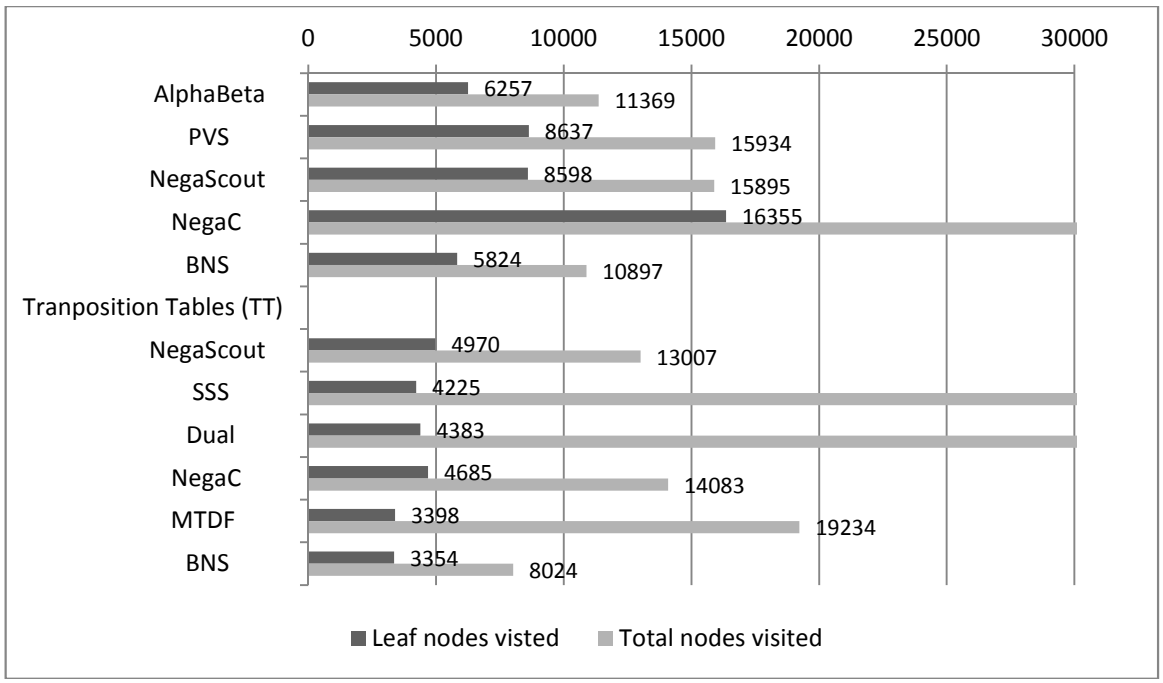


Figure 51. Tree width – 3, depth – 10

Leaf node range 0..800; full alpha-beta window

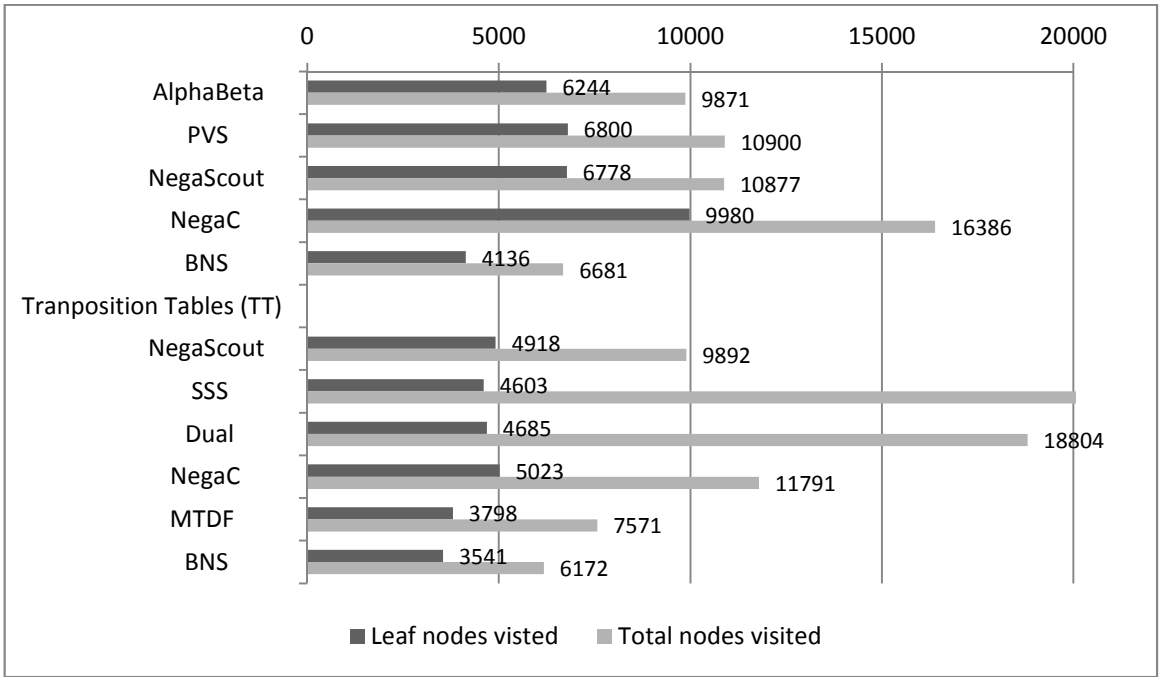


Figure 52. Tree width – 4, depth – 8

Leaf node range 0..80; full alpha-beta window

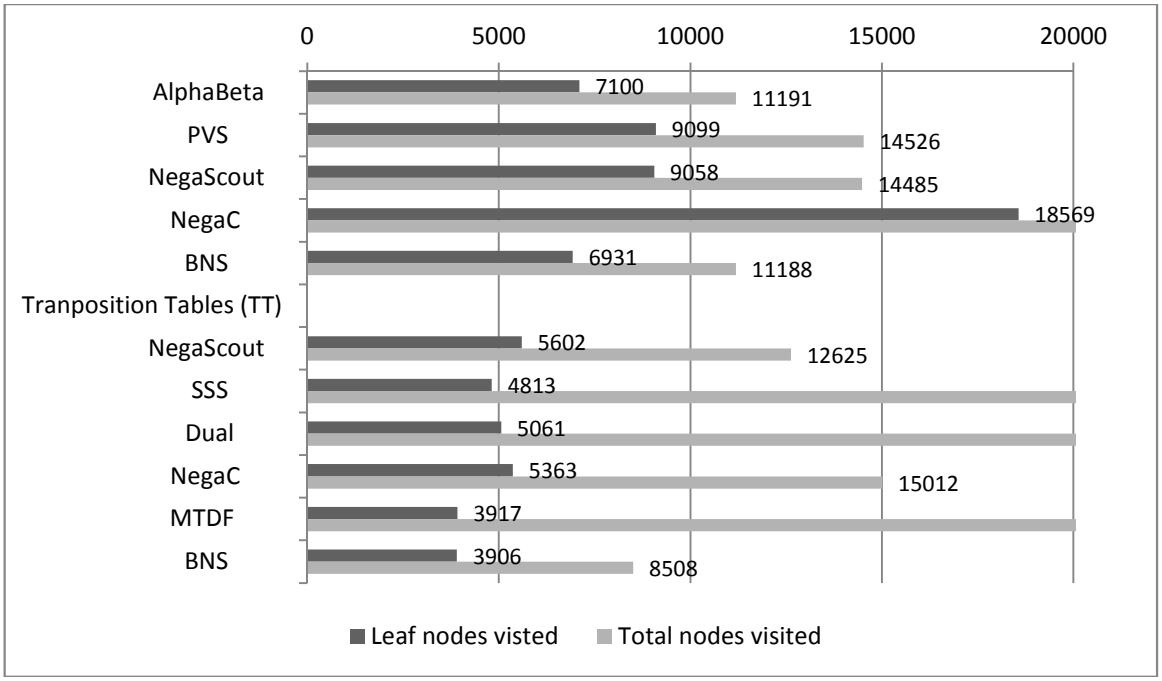


Figure 53. Tree width – 4, depth – 8

Leaf node range 0..800; full alpha-beta window

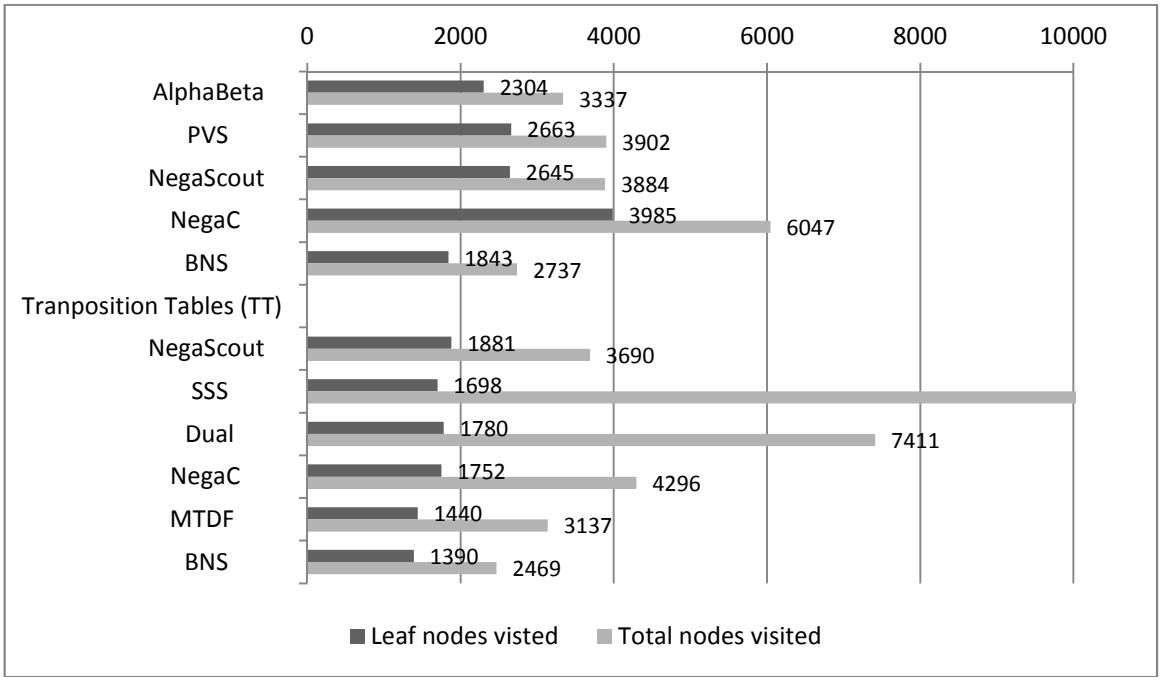


Figure 54. Tree width – 5, depth – 6

Leaf node range 0..80; full alpha-beta window

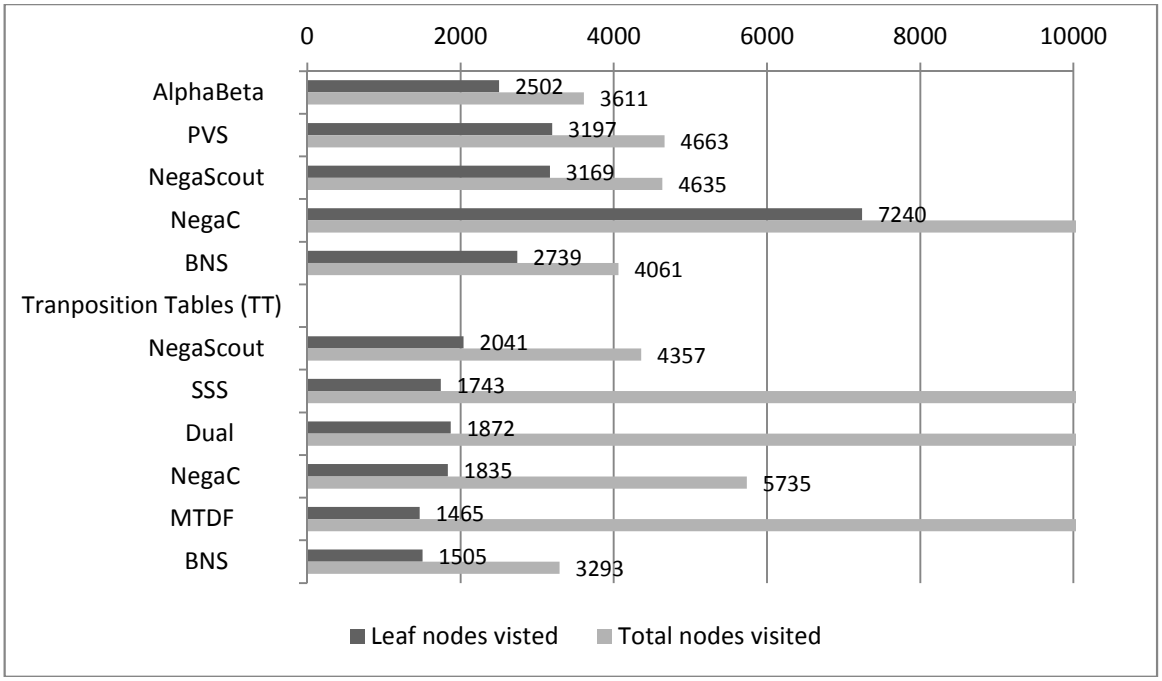


Figure 55. Tree width – 5, depth – 6

Leaf node range 0..800; full alpha-beta window

Appendix B. Experiments in Abstract Domain, Code Analysis

The following section contains the size evaluation of the program for making experiments in abstract domain. Full source code available on cd-disk attached.

```
      53 text files.
      47 unique files.
      26 files ignored.

http://cloc.sourceforge.net v 1.53  T=2.0 s (12.0 files/s, 3234.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C#                              15           1058           2563           1991
MSBuild scripts                  5              0             21            367
CSS                              1              0              0            207
XSLT                             1             28              2            202
XML                              2              0              0             29
-------------------------------------------------------------------------------
SUM:                            24           1086           2586           2796
-------------------------------------------------------------------------------
```

Appendix C. Experiments in Real Domain, Code Analysis

The following section contains the size evaluation of the program for making experiments in real domain game “Hey ! That’s my fish”. Full source code available on cd-disk attached.

```
      95 text files.
      88 unique files.
      52 files ignored.

http://cloc.sourceforge.net v 1.53  T=6.0 s (6.7 files/s, 1499.7 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C#                              30           1449           2784           3802
MSBuild scripts                  6              0             28            467
CSS                              1              0              0            207
XSLT                             1             28              2            202
XML                              2              0              0             29
-------------------------------------------------------------------------------
SUM:                            40           1477           2814           4707
-------------------------------------------------------------------------------
```

Appendix D. Experiments with Near Optimal Search, Code Analysis

The following section contains the size evaluation of the program for making experiments with near optimal search in real domain game “Hey ! That’s my fish”. Full source code available on cd-disk attached.

```
      37 text files.
      37 unique files.
      18 files ignored.

http://cloc.sourceforge.net v 1.53  T=1.0 s (19.0 files/s, 3165.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C#                              15            429            337           1804
CSS                              1              0              0            207
XSLT                             1             28              2            202
MSBuild scripts                  1              0              7            137
XML                              1              0              0             12
-------------------------------------------------------------------------------
SUM:                            19            457            346           2362
-------------------------------------------------------------------------------
```